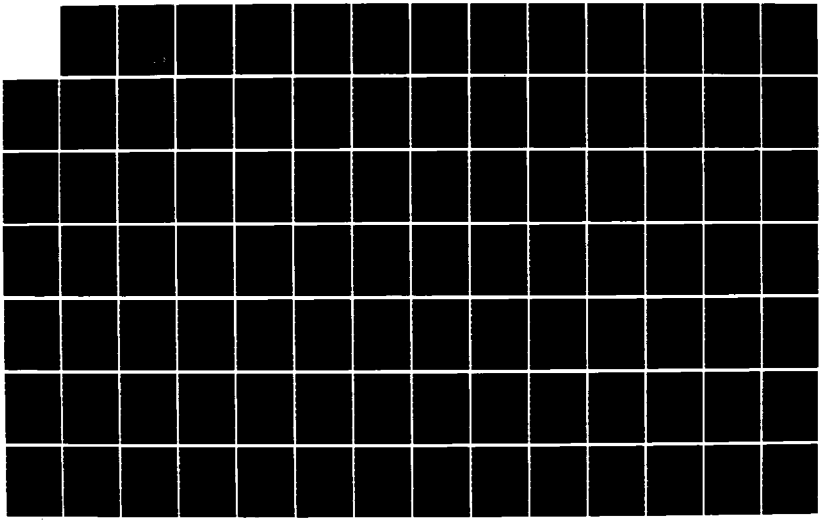
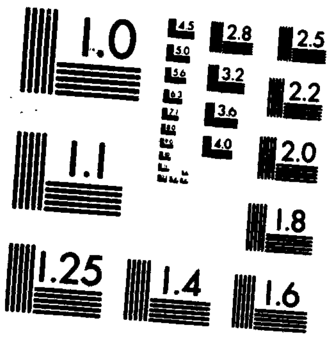




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A173 096

DTIC FILE COPY

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

①

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 86-184T	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) An Empirical Study in the Simulation of Heuristic Error Behavior		5. TYPE OF REPORT & PERIOD COVERED THESIS/~~DISSERTATION~~
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Steven R. Hansen		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Wright State University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 1986
		13. NUMBER OF PAGES 326
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

[illegible] 25 Sept 86
LYNN E. WOLAVER
Dean for Research and Professional Development
AFIT/NR

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED ...

DTIC
S ELECTE D
OCT 1 4 1986
E

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

86 10 10 094

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AN EMPIRICAL STUDY
IN THE SIMULATION OF HEURISTIC ERROR BEHAVIOR

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

STEVEN ROSS HANSEN
B.S., Brigham Young University, 1981

1986
Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

March, 1986

I hereby recommend that the thesis prepared under my supervision by Steven R. Hansen entitled An Empirical Study in the Simulation of Heuristic Error Behavior be accepted in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]
Thesis Director

[illegible]
Chairman of Department

Committee on
Final Examination

[illegible]

[illegible]

[illegible]

[illegible]

Dean of the School of Graduate Studies

QUALITY INSPECTED 4

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

ABSTRACT

Hansen, Steven R. M.S., Department of Computer Science, Wright State University, 1986. An Empirical Study in the Simulation of Heuristic Error Behavior.

Many artificial intelligence programs deal in searching for solutions from many alternatives, and depend upon heuristics to guide the search and to insure quality results. This thesis documents empirical research performed in searching problems and heuristic behavior. In this thesis A general model is presented that defines a broad class of searching domains, along with a set of software tools designed to support research in them. One puzzle configuration is devised from the model and studied in depth, examining several subtle variations of the A* searching algorithm, and results are compared with related work from other similar domains.

Heuristics are then examined as statistical entities in an attempt to substantiate theoretical work into the equivalence of heuristics, and to verify if the statistical descriptions alone are sufficient to simulate the performance of the actual heuristic. The technique of simulation by statistical profile uncovers some subtle performance trends, and promises to be a useful research tool in focusing on particular aspects of heuristic behavior.

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

TABLE OF FIGURES (CONTINUED)

TABLE OF FIGURES (CONTINUED)

TABLE OF FIGURES (CONTINUED)

TABLE OF FIGURES (CONTINUED)

LIST OF TABLES

DEDICATION

To Libby, my wife,
for her constant love and support,

and

to Kristina, Daniel, and Trieste, my children,
for their patience and understanding
during the many long hours I have been away
while involved with this effort.

I. INTRODUCTION

In the early 1800's, travel to the west coast beyond the Mississippi River was long and hazardous. No formal roads or trails existed, maps were primitive and inaccurate, supplies along the way were scarce, and many regions were inhabited by hostile indians. Before embarking on such a trek, travelers would assemble themselves into wagon trains, and would enlist the services of a guide or scout who had some knowledge of the destination and terrain along the way to help select the best, quickest and safest route to their goal. He was the one the wagon master would ask when a choice in directions was necessary. Chances were remote that the guide had seen the area in question before to know the exact answer, and instead, had to examine the clues available like the terrain and land features before making an 'educated' guess as to what the best direction might be. An incorrect choice on his part could add days to the trek, or lead the group into barren or hostile territory where the results could be fatal.

Today, Artificial Intelligence techniques are being used in an increasing number of computer applications, varying from speech recognition to molecule synthesis, robotics to expert systems. While the area of application is quite broad, virtually every AI based program uses some form of heuristic or guide to assist in selecting from among several choices or alternatives en route to a solution of the problem at hand. These heuristics are to an AI program what the guide was to the old-time wagon train, and can vary from almost perfectly educated to almost completely uninformed. The perfectly educated guide leads without deviation to the goal, while the incorrect directions supplied by the misinformed guide can lead to anything from an occasional, distracting detour to aimless meandering that never locates the destination. Just as the wagon master might have benefitted by some technique to evaluate the scouting ability of his prospective guide prior to their journey, the computer scientist could benefit by having some measures to predict the effectiveness of the heuristic guiding his program's paths.

Some investigation and research has already been conducted regarding the prediction of a heuristic's performance, inluding the effects of weighting, comparison of heuristics of differing ability, and error behavior of heuristics. Some theoretical work has been done also. Our aim was to select a new domain related to this other work, and perform empirical studies of our own. We restricted our work to the performance of heuristics using the A* search algorithm, which solves path-finding problems in strongly connected finite graphs. We hope that our results, when combined with other related research in an increasing variety of domains, will illuminate shared and common patterns, and that some encompassing theories will evolve as a result.

A. GOALS

Specifically, our goals were:

(1) to gather a significant amount of data on heuristic performance in one domain. We used a sliding-tile problem, similar to the 8-Puzzle (to be described later) for our domain.

(2) The A* algorithm we used to study heuristic behavior has two variations called Ordered Search and Graph Search. Nilsson (1980) presented the Graph Search variation and advertises it as being less redundant than Ordered Search. While his argument for Graph Search is intuitively appealing, we know of no research that provides empirical results comparing the two methods to validate his claims. We encoded both variations and ran them on a common set of data to compare the results.

(3) We wanted to compare our results directly with those gathered in other domains. Gaschnig (1979) compiled a fairly complete set of empirical studies using a similar sliding-tiles problem. We followed his methodology, used the same heuristics, and compared our results with his. Using the same heuristics, how do they perform in a different domain? What inter-search-space patterns in mean

complexity and solution quality appear?

(4) We wanted to characterize several heuristics in terms of their average statistical behavior (we call these profiles) that show the range of values the heuristic returned compared to the actual distance. This provides insight into the error behavior of the heuristic.

(5) Gaschnig (1979) claims that heuristics with identical profiles can be termed 'equivalent' and that their efficiency will be predictably the same. Using the profiles gathered in 4, we wanted to simulate classes of heuristics with duplicate behavior and review the results to verify these claims. Would their results be the same if they shared the same profile?

(6) Also using the profiles gathered in 4, we wanted to see how accurately and completely a statistical profile captured the 'intuition' of the original heuristic within the same domain. Can a heuristic be described completely using statistical performance summaries alone?

(7) Finally, we wanted to leave behind a set of tools that were general enough to be used by future researchers to gather additional data in related domains.

This document will proceed by describing the general domain we used for our research, and will show that sliding tile problems like the 8-Puzzle belong to a broad class of related puzzles of varying complexity. Our sample-gathering technique will be explained, followed by a comparison of the Ordered Search and Graph Search A* algorithm variations. We will then compare the 6-Puzzle to related work on the 8-Puzzle by other researchers. We will then describe the programs and tools we used, and show the range of puzzles over which they are designed to operate. Finally, this thesis will conclude by discussing a technique wherein the error behavior of a given heuristic is captured as a statistical 'profile', which is then used to simulate the behavior of other contrived heuristics.

A good deal of work in this thesis represents the combined efforts of Steven Hansen and Alan Cotterman, including the code used to gather the empirical results and much of the foundational aspects documented in the initial seven chapters herein. More information about this topic can be obtained in the thesis of Alan Cotterman entitled, "An Empirical Study in the Modelling of Heuristic Error Behavior".

II. BASIC SEARCH CONCEPTS

As a foundation for each of the chapters to follow, we briefly describe the search technique concepts, including definitions and algorithms which are important to our study. This chapter is intended to be a review, and excellent treatment of this subject is found in Nilsson (1980, Pp 62-88), Pearl (1984, Pp 33-34), and Gaschnig (1979, Pp 22-28).

A. TERMINOLOGY

Searching problems consist of finding a specific goal amidst a 'forest' of possibilities. This forest, formally called a state space, is generally comprised of a finite set of possible elements that can be summarized either by exhaustive enumeration of all possible configurations of the relevant objects (eg. X={1,2,3}), or by a functional description or set of transformation operators or rules that can be used to create all the elements (eg. X={n || 0 < n < 4}). The notion of a state space search implies a process that begins at an initial state and iteratively applies rules to create and move through the elements (nodes) in the state space until the goal state is reached.

The search process maintains a search graph composed of nodes characterizing the individual elements discovered within the state space. Each node consists of a description of an element's unique state or configuration. Any two nodes in the graph are connected to one another by an undirected arc only when the application of any of the rules or operators defining the state space can create one element from the other. In addition to node relationships, arcs show the cost of generating the 'neighbor' node from the original. For our purposes, cost will be uniform for all arcs and will represent simply a single state tranformation operation between nodes.

If one can begin at any given node, traverse a series of arcs, and arrive at another node, then a path is said to exist between the two, and the cost of the path is simply the number of arcs traversed between them. In a graph setting, more than one path may exist between two given nodes, and the various paths may be of different lengths (or "costs"). Naturally, when this occurs, one wishes to select the shortest (least cost) path of the set.

While the graph maintains arcs permitting any and all paths to be traced between two nodes, it doesn't keep track of which one is the shortest. For this reason, a search tree must also be maintained. A search tree is a specific case of a graph where any given node can only have one parent. The search tree connects nodes with directed arcs, pointing backwards from the generated node (successor) to a single parent node. As the search process proceeds, and multiple paths are discovered to a single successor configuration (in effect, giving that child more than one parent), it becomes necessary to select the parent (or path) with the lowest cost, redirecting the parent pointer as needed. (The multiple relationship is still maintained via undirected arcs in the graph, however.) When the search process terminates by finding the goal sought, traversing the path established in the search tree by the parent pointers gives the sequence of state transformations needed to go from the start to the goal state.

B. GENERAL SEARCH ALGORITHM

Nilsson (1980, Pp 64-65) presents an algorithm that solves searching problems in strongly connected, finite graphs. (He calls the algorithm "Graphsearch", which should not be confused with an updating variation to be examined in Chapter V called "Graph Search".) The algorithm builds the state space graph beginning with a start state as a 'seed' and systematically generates the graph (G) and search tree around it until the goal is found (success) or until all possible moves have been discovered (failure). A node is said to be expanded when all of its successors have been generated, that is, when all possible configurations one move or step away have been obtained by applying the operators mentioned above.

To control which nodes have been expanded and which remain to be, this algorithm uses two bookkeeping lists called OPEN and CLOSED. Expanded nodes are placed on the CLOSED list, while those awaiting expansion remain on the OPEN list. The algorithm iteratively removes a node from OPEN, places it on CLOSED, and expands it, placing each successor generated onto the OPEN list, repeating this sequence until it finds the goal or until OPEN is empty (i.e. no more states can be generated). The graph G collects all the paths discovered to each of the generated nodes, while the best path is shown in the search tree, using the parent pointers maintained in step 7 (listed below). When the algorithm terminates successfully (having found the goal on OPEN), the solution path can be traced backwards from that node to each of the ancestors up to the start node, giving the solution path. The algorithm follows, and is copied from Nilsson (1980).

1. Create a new search graph, G, consisting solely of the start node, s. Put s on a list called OPEN.
2. Create a list called CLOSED that is initially empty.
3. If OPEN is empty, exit with failure.
4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this node n.
5. If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G. (Pointers are established in step 7.)
6. Expand node n, generating the set M of its successors and install them as successors of n in G.
7. Establish a pointer to n from those members of M that were not already in G (i.e. not already

on either OPEN or CLOSED). Add these members of *M* to *OPEN*. *For each member of* M that was already on OPEN or CLOSED, decide whether or not to redirect its pointer to n. For each member of M already on CLOSED, decide for each of its descendands in G whether or not to redirect its pointer. (Chapter V will treat this step in detail)

8. Reorder the list OPEN, either according to some arbitrary scheme or according to heuristic merit.

9. Go to step 3.

1. DISCUSSION

The algorithm is self-expanatory except for steps 7 and 8, which can have several interpretations and variations. Step 7 handles multiple paths to a single node by first recognizing that a newly generated successor has been 'seen' *before (and hence has more than one parent)*. This is done by comparing each successor with every entry on OPEN and CLOSED. A duplicate indicates that another parent exists for this configuration and that the new node is redundant and must be discarded. The rediscovered node keeps both parents as neighbors in the search graph, but the search tree forces the child to 'choose' one of the two. The child decides by pointing to the parent with the shortest path to the root. This is called updating a node, and there are two alternative methods which accomplish this, called Ordered Search and Graph Search. These variations are discussed in detail in Chapter V.

2. REORDERING VARIATIONS

Step 8 refers to reordering the nodes on OPEN. Nodes are typically ordered based on their cost, or distance from the start node. Several alternative ordering methods are possible, and the choice can drastically affect the direction and efficiency of the resulting search pattern. Four variations are presented below, called Depth-First, Breadth-First, A*, and weighted A*.

a. DEPTH-FIRST SEARCH

One variation is called Depth-First Search, which is characterized by a search pattern that proceeds downward along a single path until (1) the goal node is found, (2) a node on the existing path cannot be expanded further (called a 'terminal' node), or (3) some arbitrary depth bound is reached. If the goal is found, then the algorithm succeeds and terminates with the solution path. Otherwise, it backs up one level, selects the most promising alternative and proceeds downward again. The OPEN list in this variation is kept in descending order by cost, where cost is the depth of the node from the start.

b. BREADTH-FIRST SEARCH

Another variation, called Breadth-First Search, expands the graph completely at each level before advancing to the next deeper level. The OPEN list in this case is maintained in ascending order by the depth of the node from the start (same cost measure as used in Depth-First).

Breadth-First guarantees to find a solution if one exists, and that the one it finds will be the shortest one in the graph (this property is referred to as 'admissibility'). This good feature is offset by the cost incurred in exhaustively enumerating the state space layer by layer up to the level in which the goal resides.

c. A* SEARCH

Another variation, called A*, attempts to combine the good effects of both of the above through use of heuristics to guide the direction of the search pattern by intelligently ordering OPEN. Informally, a heuristic is a 'rule of thumb', an educated guess, or intuition applied to the task at hand. Heuristics provide a simple means of indicating which among several courses of action is to be preferred, but are not guaranteed to identify the most effective course of action. Obviously, the more accurate and consistent a heuristic is, the more effective it is and the more efficient the resulting version of A* becomes.

The cost of a node is computed by using the following formula:

$$F(n) = G(n) + H(n)$$

where n is the node, G is the distance of node n from the start node (same measure as in Depth and Breadth First), and H is a heuristic estimate of the distance (cost) remaining to the goal. F(n) is then the program's best estimate of the solution path length for a path constrained to pass through node n. The OPEN list is kept in ascending order on F.

The heuristic component guides the directions in which the search tree is developed, discouraging Breadth-First expansion and permitting the program to expand nodes along paths it senses (sometimes incorrectly) are the way to the goal. Without the H component, A* reduces to Breadth-first search, and without the G component, A* relies purely on the estimating ability of the heuristic. This is all right if the heuristic is accurate or if many paths exist to the goal, but can lead to long searches down dead-end paths if H occasionally returns misleading values. The G-component serves to remind the program that the search has been (or is being) led astray.

In addition, if the H component always underestimates the actual distance remaining to reach the goal, the property of admissibility is retained, and A* will always find the shortest path from the starting state to the goal state.

d. WEIGHTED A*

Pohl (1970) modified the A* algorithm by using weights to adjust the effect of the two components:

$$F(n) = (1-W) * G(n) \quad + \quad W * H(n)$$

In this function, G and H mean the same as they do in A*, but the percentage of their contribution to the cost measure F is controlled through the selection of a weight W between 0 and 1. Note that setting the weight to zero reduces to Breadth-first search, and a weight of one ignores the G component completely, resulting in a purely heuristic search. Using a weight of one-half evenly balances G and H, corresponding to the classical A* algorithm above.

Because the use of weights provides such a variety of cost-ordering variations, we chose to use Weighted A* for all of the work presented in this thesis. We will discuss this algorithm further in Chapter V, where we explore various updating mechanisms.

III. BEADS WORLD

This chapter describes the domain (or problem model) we selected with which to conduct our research. We will relate our methodology to that of other AI research, define our model, and show that it is part of a large family of related models.

A. NEED FOR RESEARCH MODELS

"Our research strategy in studying complex systems is to specify them in detail, program them for digital computers, and study their behaviour empirically by running them with a number of variations and under a variety of conditions. This appears at present the only adequate means to obtain a thorough understanding of their behaviour." (Newell, Shaw, and Simon, 1963)

A great deal of AI research to date has been conducted using games or puzzles for the problem domain. We did the same in this thesis. We (AI researchers) are not fascinated with games and puzzles any more than geneticists are enamored with fruit flies. Each simply provides an experimental guinea pig to the researcher that is easy to define in detail, yet whose behavior is sufficiently rich and unpredictable to simulate the complexity found in real-life situations. Most real-life situations are too irregular and complex to describe concisely, and in sufficient detail for fellow researchers to comprehend, much less program for computer execution.

B. THE 8-PUZZLE

Gaschnig (1979) selected a sliding-tiles problem called the 8-Puzzle for part of his dissertation research because "it is a simple yet non-trivial case study in which to explore general issues with rigor, principally the issue of predicting algorithm performance." (Gaschnig, 1979, Pg 3) The 8-puzzle, a game still sold in many toy stores, consists of eight numbered, movable square tiles placed on a 3 X 3 matrix, with the ninth matrix element left blank or unoccupied. Having this empty cell in the matrix makes it possible for any orthagonally adjacent numbered tile to move into its place, allowing the configuration of numbered tiles to change to over 180,000 different permutations. By arbitrarily selecting one of these permutations of numbered tiles as a "starting state", a carefully chosen sequence of tile movements about the matrix will transform this initial configuration into a preselected goal permutation. The basic objective is not only to maneuver the tiles so as to reach the goal state, but also to do so in as few moves as possible. Here is an example of part of the 8-Puzzle state space:

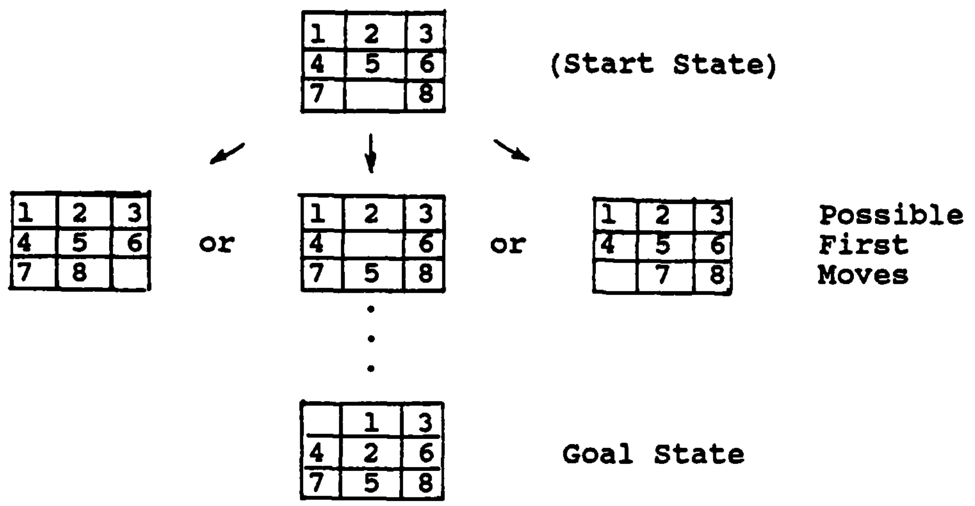


Figure 3.1
A Portion of the 8-Puzzle State Space

C. GENERALIZING SLIDING-TILE PUZZLES

The description of the 8-Puzzle above is virtually identical to the ones provided by Gaschnig (1979, Pp 22-23), Pearl (1984, Pp 6-7), and Nilsson (1980, Pp 18-20). However, an alternate method of characterizing the 8-Puzzle is to say that it has a connected ring of eight positions surrounding a single center, and that positions are occupied by one of eight numbered, mobile markers or "beads". Markers are only permitted to move (1) between adjacent ring positions, and (2) between the center and every alternate ring position. Of course, the objective is still to rearrange an initial starting configuration into a preselected goal configuration in as few moves as possible. The figure below depicts three equivalent instantiations based on this definition.

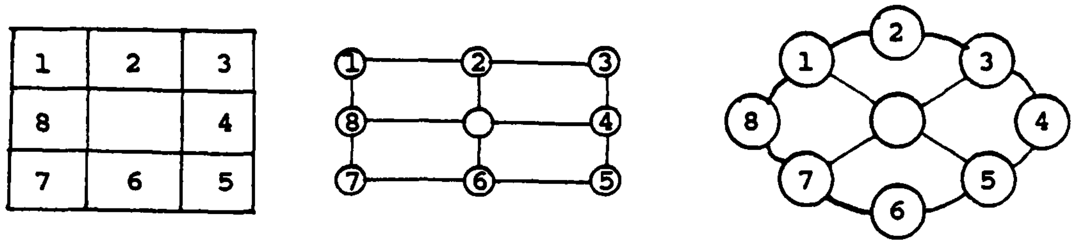


Figure 3.2
Equivalent 8-Puzzle Instantiations

Defining the 8-Puzzle in this way relieves it of the matrix constraint and makes its ultimate shape irrelevant. We can characterize it completely in terms of its number of positions and by the set of legal moves or transformations (defined by which positions are inter-connected to allow bead movement). We can carry the abstraction one step further by lifting the restriction on the number of positions in the perimeter ring. One is no longer constrained to eight positions as in the 8-Puzzle and could create a different variation with only five or ten positions in the outer ring.

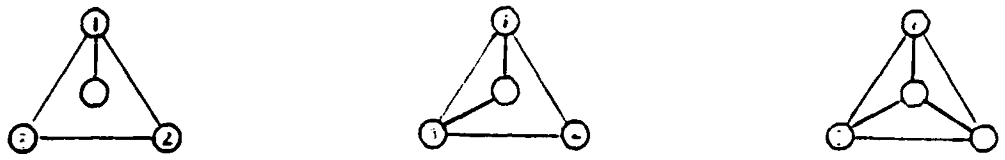
An additional generalization is to lift the restriction on every alternate position being linked to the center and allow this to be defined in whatever number and configuration is desired. Such an extension enables augmenting the orthagonal moves in the 8-Puzzle with diagonal moves, for example. By using these abstractions and varying the number of positions and/or changing the configuration of center-perimeter links, an entire class of 8-Puzzle mutations can be devised, each sharing the 8-Puzzle's objectives but displaying a variety of behavior.

1. BEADS WORLD DEFINITION

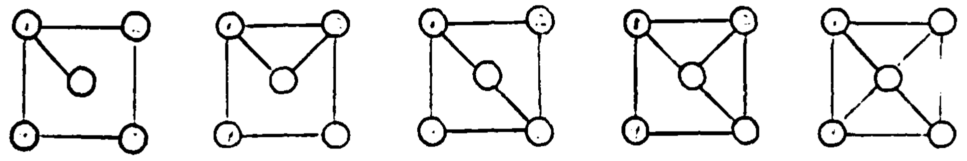
The abstraction provided above gives the basic framework for defining a class of puzzles similar to the 8-Puzzle. We refer to this class as the 'Beads World'. Essentially, the Beads World is composed of puzzles characterized by a set of positions linked into a ring situated around a single center position. Numbered beads (or markers) occupy all but one of these positions. The number of beads used, incidentally, is what gives the puzzle its name. The vacant or blank position is necessary to allow the beads room to move about. Beads are permitted to move from position to position when two conditions exist: (1) a path or link has been established between the two positions, and (2) the destination position is blank or unoccupied. Since by definition, all perimeter positions are connected into a ring-like structure, movement between adjacent perimeter positions is automatically allowed. In addition, any combination of perimeter positions may be defined as being linked to the center, so long as at least one is. The 8-Puzzle, then, has 9 positions, 8 beads, and every other perimeter position has a path to the center.

Note that it is up to the user to set the number of positions and to establish which perimeter positions have a path or link to the center. Altering the number of positions affects the size of the state space of the puzzle. Changing the number and/or the configuration of center-to-perimeter links redefines the rules or operators that create the nodes in the state space, which also affects the shape and size of the state space. Examples of some of the many possible puzzle permutations are shown in Figure 3.3.

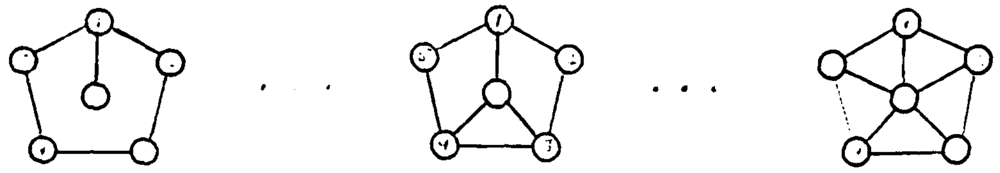
FIGURE 3.3
Beads World Variations



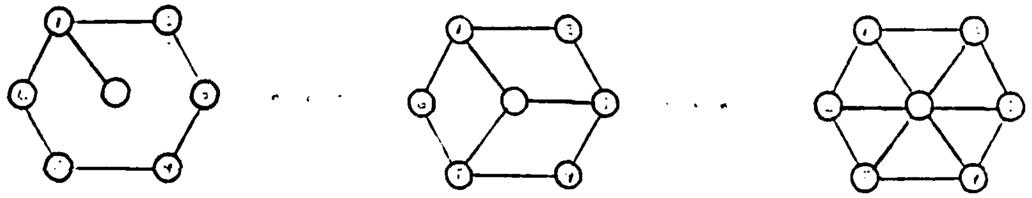
3-Puzzle Family



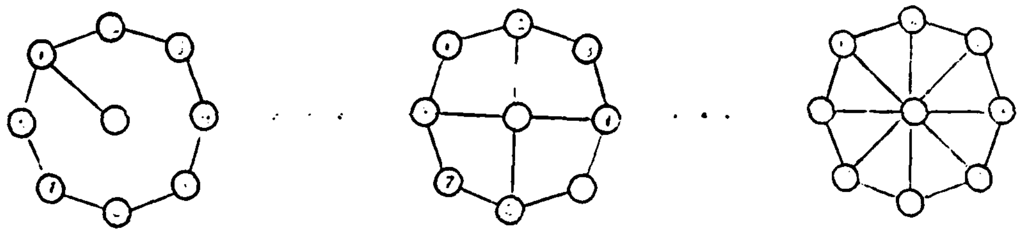
4-Puzzle Family



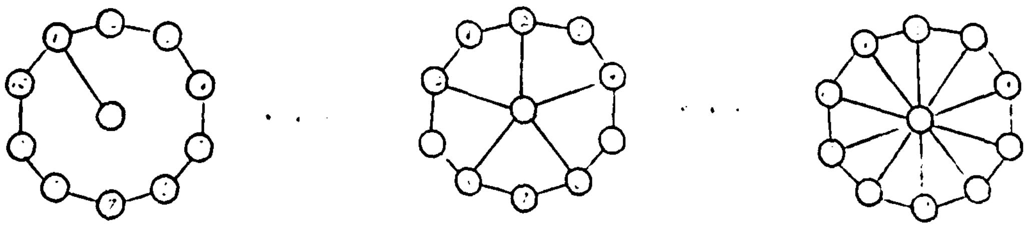
5-Puzzle Family Members



6-Puzzle Family Members



8-Puzzle Family Members



10-Puzzle Family Members

2. EXTENDING THE GENERALIZATION

So far, this discussion has mainly focused on generalizing sliding-tile (or beads) problems, which we have illustrated is simple to do and provides a wealth of related family members with which to experiment. Our programs function with any of the members defined thus far. However, this generalization can be extended even further to encompass an even wider class of problems.

We envision a class of puzzles using 'baling wire' and beads, where the wire determines the paths that the beads may travel. The beads may or may not be marked. Positions do not need to be connected in a ring, nor is a single center position required. In fact, by creating a separate ring structure centered within another ring structure, with wires connecting the two, we create a family of problems that encompasses the 15-Puzzle and all its relatives.

It is even possible to mimick the blocks world within our Beads World generalization. The blocks world consists of N numbered (or colored) cubes which may be arranged into various stacks on a table (Nilsson, 1980, Pg 152). It is often used to illustrate AI planning and searching algorithms. By abandoning the ring shape of our Beads World model, one could devise the blocks world problems from the Beads World definition.

All one needs to do to define his 'Beads World' puzzle is (1) determine a basic shape (ring around a center, ring around a ring, three-dimensional matrix, etc), (2) determine the number of available positions, (3) determine the number of beads, and (4) establish the paths the beads will traverse (or how to connect the wires up). The full extent of the Beads World family of models has not been explored, and we leave this as an idea for further development.

D. EXAMINATION OF THE 6-PUZZLE FAMILY

To illustrate the potential and versatility of the puzzles that this model defines (and our software tools can manipulate), we present the 6-Puzzle (consisting of seven positions and six beads) in all of its possible link permutations. Family members were generated by systematically altering the number and position of links from the perimeter slots to the center, creating twelve unique non-isomorphic configurations. For each of the twelve, we used a common starting state (shown below) and generated its state space.

```
   1-----2
  /    /  \
 6----0    3
  \    \  /
   5-----4
```

In the pages that follow, Tables 3.1 through 3.12 highlight the results of the twelve variations, including a diagram of the puzzle showing the links used, a histogram showing the relative shape of the search tree, and figures indicating the longest sequence of moves discovered (maximum depth of the tree), the number of possible states reached, and the branching factor (average number of successors from a given parent).

TABLE 3.1
6-PUZZLE CONFIGURATION 1

```
  o-----o
 /       \
o----o    o             Unique States        :    35
 \       /              Avg # of Neighbors   :     2.00
  o-----o               Maximum Depth        :    16


                Nodes at each level:

                   1 --      1  *
                   2 --      2  **
                   3 --      2  **
                   4 --      2  **
                   5 --      2  **
                   6 --      2  **
                   7 --      2  **
                   8 --      4  ****
                   9 --      2  **
                  10 --      2  **
                  11 --      2  **
                  12 --      2  **
                  13 --      2  **
                  14 --      4  ****
                  15 --      2  **
                  16 --      1  *
```

TABLE 3.2
6-PUZZLE CONFIGURATION 2

```
   o-----o
  / \     \
 o----o    o          Unique States        :  5040
  \       /           Avg # of Neighbors   :     2.29
   o-----o            Maximum Depth        :    63
```

Nodes at each level:
(Even levels removed for brevity)

```
 1 --     2
 3 --     5
 5 --     4
 7 --     6
 9 --    12
11 --    12
13 --    14
15 --    28  *
17 --    32  *
19 --    32  *
21 --    49  *
23 --    77  **
25 --    72  *
27 --    86  **
29 --   148  ***
31 --   156  ***
33 --   144  ***
35 --   222  ****
37 --   276  *****
39 --   225  ****
41 --   233  *****
43 --   294  ******
45 --   182  ****
47 --   106  **
49 --    48  *
51 --    18
53 --    14
55 --    10
57 --     4
59 --     4
61 --     4
63 --     1
```

TABLE 3.3
6-PUZZLE CONFIGURATION 3

```
    o-----o
   /    /  \
  o----o    o          Unique States        :  2520
   \       /           Avg # of Neighbors   :     2.29
    o-----o            Maximum Depth        :    42

                   Nodes at each level:
                      1 --     2
                      2 --     4
                      3 --     4
                      4 --     6
                      5 --     6
                      6 --     9
                      7 --     8
                      8 --    14  *
                      9 --    14  *
                     10 --    24  *
                     11 --    22  *
                     12 --    34  *
                     13 --    32  *
                     14 --    52  **
                     15 --    49  **
                     16 --    78  ***
                     17 --    68  ***
                     18 --    99  ****
                     19 --    83  ***
                     20 --   118  *****
                     21 --   102  ****
                     22 --   160  ******
                     23 --   139  ******
                     24 --   194  ********
                     25 --   136  *****
                     26 --   162  ******
                     27 --   127  *****
                     28 --   177  *******
                     29 --   130  *****
                     30 --   169  *******
                     31 --    95  ****
                     32 --    93  ****
                     33 --    43  **
                     34 --    26  *
                     35 --     8
                     36 --     8
                     37 --     6
                     38 --     7
                     39 --     4
                     40 --     4
                     41 --     2
                     42 --     1
```

TABLE 3.4
6-PUZZLE CONFIGURATION 4

```
   o-----o
  /       \
 o----o----o          Unique States          :   840
  \       /           Avg # of Neighbors     :     2.29
   o-----o            Maximum Depth          :    29
```

```
Nodes at each level:

     1 --     2
     2 --     4
     3 --     4
     4 --     4
     5 --     8  *
     6 --     8  *
     7 --    12  *
     8 --    16  **
     9 --    16  **
    10 --    26  ***
    11 --    28  ***
    12 --    32  ****
    13 --    52  ******
    14 --    52  ******
    15 --    66  ********
    16 --    88  **********
    17 --    79  *********
    18 --    92  ***********
    19 --    78  *********
    20 --    52  ******
    21 --    48  ******
    22 --    24  ***
    23 --    16  **
    24 --    14  **
    25 --     6  *
    26 --     5  *
    27 --     4
    28 --     2
    29 --     1
```

TABLE 3.5
6-PUZZLE CONFIGURATION 5

```
  o-----o
 /  \ /  \
o----o    o
 \       /
  o-----o
```

```
Unique States         :  5040
Avg # of Neighbors    :     2.57
Maximum Depth         :    43
```

Nodes at each level:
(Even levels removed for brevity)

```
 1 --     3
 3 --     8
 5 --    12
 7 --    17
 9 --    28  *
11 --    46  *
13 --    66  *
15 --   101  **
17 --   146  ***
19 --   182  ****
21 --   220  ****
23 --   299  ******
25 --   330  *******
27 --   289  ******
29 --   307  ******
31 --   264  *****
33 --   136  ***
35 --    34  *
37 --    14
39 --    11
41 --     6
43 --     1
```

TABLE 3.6
6-PUZZLE CONFIGURATION 6

```
   o------o
  /     /  \
o-----o-----o          Unique States        :  5040
 \         /           Avg # of Neighbors   :     2.57
   o------o            Maximum Depth        :    29
```

Nodes at each level:

```
 1 --     3
 2 --     6
 3 --     7
 4 --     9
 5 --    16
 6 --    24
 7 --    35  *
 8 --    50  *
 9 --    74  *
10 --   109  **
11 --   149  ***
12 --   203  ****
13 --   278  ******
14 --   348  *******
15 --   409  ********
16 --   495  **********
17 --   585  ************
18 --   616  ************
19 --   530  ***********
20 --   383  ********
21 --   265  *****
22 --   175  ***
23 --   105  **
24 --    67  *
25 --    48  *
26 --    28  *
27 --    15
28 --     6
29 --     1
```

TABLE 3.7
6-PUZZLE CONFIGURATION 7

```
  o------o
 /    /  \
o----o    o          Unique States        :  2520
 \    \  /           Avg # of Neighbors   :     2.57
  o------o           Maximum Depth        :    20
```

Nodes at each level:

```
 1 --     3
 2 --     6
 3 --     6
 4 --    12
 5 --    18   *
 6 --    33   *
 7 --    36   *
 8 --    72   ***
 9 --    90   ****
10 --   168   *******
11 --   186   *******
12 --   329   *************
13 --   321   *************
14 --   452   ******************
15 --   297   ************
16 --   293   ************
17 --   108   ****
18 --    69   ***
19 --    15   *
20 --     5
```

TABLE 3.8
6-PUZZLE CONFIGURATION 8

```
  o-----o
 /  \ /  \
o----o----o               Unique States        :  5040
 \       /                Avg # of Neighbors   :     2.86
  o-----o                 Maximum Depth        :    27

                      Nodes at each level:

                          1 --     4
                          2 --     8
                          3 --    11
                          4 --    18
                          5 --    32  *
                          6 --    49  *
                          7 --    64  *
                          8 --    77  **
                          9 --   101  **
                         10 --   139  ***
                         11 --   190  ****
                         12 --   253  *****
                         13 --   319  ******
                         14 --   398  ********
                         15 --   486  **********
                         16 --   574  ***********
                         17 --   617  ************
                         18 --   547  ***********
                         19 --   415  ********
                         20 --   285  ******
                         21 --   180  ****
                         22 --   111  **
                         23 --    71  *
                         24 --    47  *
                         25 --    26  *
                         26 --    13
                         27 --     4
```

TABLE 3.9
6-PUZZLE CONFIGURATION 9

```
     o-----o
    /     / \
   o----o----o          Unique States       :  5040
    \     \ /           Avg # of Neighbors  :     2.86
     o-----o            Maximum Depth       :    21
```

Nodes at each level:

```
 1 --     4
 2 --     8
 3 --    10
 4 --    18
 5 --    36  *
 6 --    61  *
 7 --    87  **
 8 --   134  ***
 9 --   213  ****
10 --   339  *******
11 --   488  **********
12 --   666  *************
13 --   813  ****************
14 --   858  *****************
15 --   687  **************
16 --   388  ********
17 --   167  ***
18 --    43  *
19 --    14
20 --     4
21 --     1
```

TABLE 3.10
6-PUZZLE CONFIGURATION 10

```
    o-----o
   /  \ /  \
  o    o    o          Unique States       :  5040
   \  / \  /           Avg # of Neighbors  :     2.86
    o-----o            Maximum Depth       :    21

                    Nodes at each level:

                        1 --     4
                        2 --     8
                        3 --    10
                        4 --    16
                        5 --    36  *
                        6 --    63  *
                        7 --    94  **
                        8 --   136  ***
                        9 --   216  ****
                       10 --   358  *******
                       11 --   514  **********
                       12 --   668  *************
                       13 --   808  ****************
                       14 --   815  ****************
                       15 --   638  *************
                       16 --   376  *******
                       17 --   168  ***
                       18 --    75  *
                       19 --    31  *
                       20 --     4
                       21 --     1
```

TABLE 3.11
6-PUZZLE CONFIGURATION 11

```
    o-----o
   /  \ /  \
  o----o----o            Unique States          :  5040
   \    \  /             Avg # of Neighbors     :     3.14
    o-----o              Maximum Depth          :    18
```

Nodes at each level:

```
 1 --     5
 2 --    10
 3 --    14
 4 --    30  *
 5 --    68  *
 6 --   119  **
 7 --   174  ***
 8 --   236  *****
 9 --   362  *******
10 --   561  ***********
11 --   738  ***************
12 --   858  *****************
13 --   835  *****************
14 --   625  ************
15 --   313  ******
16 --    79  **
17 --    11
18 --     1
```

TABLE 3.12
6-PUZZLE CONFIGURATION 12

```
   o-----o
  / \ / \
 o----o----o          Unique States        :  5040
  \ / \ /             Avg # of Neighbors   :     3.43
   o-----o            Maximum Depth        :    15

                  Nodes at each level:

                      1 --     6
                      2 --    12
                      3 --    18
                      4 --    48  *
                      5 --   120  **
                      6 --   219  ****
                      7 --   338  *******
                      8 --   438  *********
                      9 --   647  *************
                     10 --   948  *******************
                     11 --   930  ******************
                     12 --   660  *************
                     13 --   427  ********
                     14 --   194  ****
                     15 --    34  *
```

While it is interesting to compare the relative shapes of the trees, we found two items most intriguing and worthy of further investigation. First is the range of tree depths, which varied from 15 to 63 levels (or moves). Also of interest was the size of the state space generated in comparison to the total number of combinations possible. Some configurations could not reach all the possible permutations of moves. There are 7! or 5040 possible unique states, and only configurations 2, 5, 6, and 8 through 12 reached them all. Configurations 3 and 7 only reached 2520 states, or 1/2 of the total possible. Puzzle 4 only managed 840 states, or 1/6 of the number possible, and puzzle 1 only created 35 states, which is 1/144 of the number possible. There seems to be a relationship between the number of moves in the shortest cycle to the number of states that puzzle can reach, where a cycle is the shortest series of moves required to go from the center position out to the perimeter and return without retracing any paths previously traversed. Those puzzles reaching all 5040 nodes had a minimal cycle of 3 moves, while the other puzzles reached subsets that were inversely porportional to the length of their minimal cycles. It would be interesting to mathematically define this relationship to enable the prediction of this value.

E. SUMMARY

The beauty of the Beads World generalization is that the researcher may now vary this well-defined model and tune it to meet specific goals or to examine specific behavior. This is a luxury not as easily enjoyed by the geneticist and his fruitfly. Generally speaking, we observe that adding links increases the number of puzzle state permutations in the state space, increases the branching factor, and decreases the tree depth. Of course, adding positions increases the magnitude of a puzzle dramatically: the 6-Puzzle has 7! or 5040 states, the 7-Puzzle has 8! or 40,320 states, the 8-Puzzle has 362,880 states, and so on.

For our purposes, we sought a puzzle that was close to the 8-Puzzle in behavior, yet smaller so that exhaustive enumeration would be possible on a timely basis. The characteristics of the 8-Puzzle are (1) it is symmetrical, (2) every other perimeter position is linked to the center, and (3) it possesses a bifurcated state space. We ruled out any of the 7-Puzzle family because they were assymetrical. The overall size of the 6-Puzzle was large enough to be interesting but small enough to be manageable, and it possessed a symmetrical shape. In examining the various configurations of the 6-Puzzle family, we observed that both configurations 3 and 7 subdivided the state space into two components. However, in configuration 3 (see Table 3.3), every other perimeter position was not linked to the center, whereas in configuration 7 (see Table 3.7) they were. Therefore, on the basis of puzzle symmetry, state space decomposition pattern, and link similarity to the 8-Puzzle, we selected configuration 7 as the model for the empirical studies in the remainder of this thesis, and all remaining references to the 6-Puzzle refer to this particular configuration rather than the general family.

Our work represents a fairly exhaustive treatment of only one configuration of the 6-Puzzle family, and using our tools, further research could be continued into other Beads World configurations to gather a wider base of empirical data on which to examine the behavior of search problems using heuristics.

IV. SAMPLE GENERATION PROCESS

Before continuing with the chapters detailing the results of our empirical work, we wish to pause briefly and generally describe the data sample that was used and how it was created.

There are over 6 million start/goal pairs in the 6-Puzzle, and to solve each of them to gather our figures was clearly out of the question. Instead, we needed to select a reasonable subset with which to work. This sample needed to consist of a set of start/goal configuration pairs whose solution path lengths varied from one to twenty (the maximum depth of the 6-Puzzle) depending on the particular pair. For each start state, our program attempts to find a path to the designated goal, and we keep aggregate statistics for each level in a variety of categories based on the entire sample. These statistics form the data that is examined in the subsequent chapters of this thesis; therefore, it is important that the sample we used be a fair representation of the 6-Puzzle.

A. GASCHNIG'S METHODS

Gaschnig (1979, Pp 47-51) was presented with the same problem, and used two methods to gather his samples (the second sample was generated because his first sample was accused of being biased). The first method entailed selecting a starting configuration and randomly applying the transformation operations N times, creating a candidate goal configuration. To ensure that N represented the shortest path between the pair, the A* algorithm was used on the candidate pair since A* guarantees that it will find the shortest path between start and goal if such a path exists. If the resulting solution path discovered and N were equal, the pair was added to the sample. This was repeated to gather forty pairs at each N, which for the 8-Puzzle range from one to thirty. There were three problems with this technique: (1) it required many executions of A*; (2) many candidate pairs were rejected because random application of the tranformation rules did not prevent loops and detours from occurring; and (3) finding samples at N greater than 25 was like "finding the needle in the haystack", and the rejection rate was so high that he decreased his sample sizes to only eight entries at level 28 and none at levels 29 and 30.

The other method involved selecting a start state and a random permutation of the numbers 0 through 9 as the candidate goal configuration. A* was used on each candidate pair to determine if the pair was solvable (remember that the state space is bifurcated, and the start might be in one component while the goal is in the other), and if so, what the solution path length (N) was. The problems he encountered with this method included (1) many A* executions, and (2) it was difficult to control the number of start/goal pairs found at a given N. As in his first method, the sample size tapers off at higher values of N.

B. OUR METHOD

We could have followed either of the techniques used by Gaschnig in the creation of his samples, but we chose not to for 3 reasons: (1) his methods were computationally expensive, (2) it was difficult to control the size of the sample at each value N, and (3) the 6-Puzzle's smaller state space gave us an option Gaschnig didn't have -- we could simply build the entire state space from a given start configuration (using the same process that created the tables in Chapter III), and carefully select our sample from the resulting search tree. This tree shows not only the path from the start to a possible 2520 candidate goal states, but also provides the actual distance between the pair.

While Gashnig chose a fixed number of samples for each number of moves (N) from the goal, we gathered a varying number of states at each level of the search tree to form our sample. This number was comprised of a pre-determined minimum (we used the number 5) and an additional amount representing proportionately the number of nodes at that level compared to the total number of nodes in the tree.

This permitted the sample to be 'shaped' as the graph itself was, giving a greater number of samples on those levels containing the greatest number of possibilities. It also provided an absolute minimum to select at those levels where relatively few nodes exist. Gathering the nodes was a simple matter of building the search tree and randomly selecting a proportional number of 'goal' nodes from each level. In addition to outputting the start and goal configurations, we also printed the actual distance between them since some of the programs later on needed this information. This saved the expense of recalculating the minimum distance again later.

Using a minimum of 5 samples per level (assuming there were at least five to choose from), our program generated a total of 198 start/goal pairs. The table below summarizes the number of goal node puzzle states taken from each level of the search tree. Note that our sample represents 198 out of 6 million possible combinations, or a selection ratio of 1 in 30,000. Gashnig's sample contained 895 of 60 billion possible combinations, for a selection ratio of 1 in 600 million!! Therefore, our sample is several orders of magnitude more complete than his was.

TABLE 4.1
SAMPLE SUMMARY

level	Nodes at Level	Number selected
1	3	3
2	6	5
3	6	5
4	12	5
5	18	6
6	33	6
7	36	6
8	72	8
9	90	9
10	168	12
11	186	12
12	329	18
13	321	18
14	452	23
15	297	17
16	293	17
17	108	9
18	69	8
19	15	6
20	5	5
	------	----
	2520	198

V. COMPARISON OF ORDERED SEARCH AND GRAPH SEARCH

The control mechanism used by many programs to solve searching problems in strongly connected, finite graphs is called the A* Algorithm. This procedure provides methodical, efficient means of expanding nodes in a graph setting until a goal is found (if one exists). This chapter builds upon the introduction provided in Chapter II, focusing primarily on the A* algorithm variations that deal with nodes that are rediscovered during the search process. Nilsson (1980) presented a variation we called Graph Search which is advertised as more efficient than the prevailing method called Ordered Search. We first discuss the Ordered Search strategy, which seems to be more commonly used, followed by Graph Search, and illustrate both with examples. We then present the results of our empirical comparison.

A. OVERVIEW

As a review, in a graph setting, multiple paths can exist to any single puzzle state. Granted, the primary objective of the search is to find any path to the goal; but when several paths lead to the same node, why not weed out the longer ones in favor of the shortest and most direct one? Both Ordered Search and Graph Search do this, but their methods are distinct and involve tradeoffs in the computation time and space required, and in the number of nodes expanded. Simply stated, the Ordered Search variation maintains only a search tree and not the graph, returning nodes that are rediscovered but at a lower cost back onto OPEN for possible reexpansion later. Note the inherent redundancy since the same node can be rediscovered and possibly reexpanded several times. However, it does save the time and space needed to keep a graph structure current.

On the other hand, Graph Search maintains both the search tree and a sub-graph. The tree shows the least-cost path to the root via parent pointers (just as in Ordered Search). The graph keeps track of the 'neighborhood', or every path to every node discovered thus far. Cheaper paths to existing nodes are maintained by propogating the new path information to the neighbors of the affected node (as kept by the neighbor pointers in the graph), redirecting parent pointers in the search tree as needed. Propogating values through the subgraph in this manner eliminates the need to ever re-expand nodes. This savings is offset, however, by the overhead required to maintain the graph.

> "There is a tradeoff between the computational cost of [maintaining the graph structure] and conputational cost of [re-expanding rediscovered nodes]" (Nilsson, 1980, Pg 66)

"Nilsson's variation saves reexpansion effort at expense of value propogation and pointer redirecting effort..." (Pearl, 1984, Pg 49)

B. ORDERED SEARCH ALGORITHM

Here is the Ordered Search variation of the A* algorithm, adapted from the A* algorithm discussed in Chapter II of this thesis. Note the differences in steps 1 and 7, which will be discussed momentarily.

```
1. Create a new search tree, T, consisting
   solely of the start node, s.  Put s on a list
   called OPEN.

2. Create a list called CLOSED that is initially
   empty.

3. If OPEN is empty, exit with failure.

4. Select the first node on OPEN, remove it from
   OPEN, and put it on CLOSED.  Call this node n.

5. If n is a goal node, exit successfully with the
   solution obtained by tracing a path along the
   pointers from n to s in T. (Pointers are
   established in step 7.)

6. Expand node n, generating the set M of its
   successors.

7. For each node m in M do the following:
   (A) If m not on OPEN or CLOSED then
          establish n as the parent of m,
          add m to OPEN
   (B) If m on OPEN
        if cost of new m < old m
            remove old m from OPEN,
            discard old m,
            make n the parent of new m,
            add new m to OPEN.
        otherwise
            ignore new m
   (C) If m on CLOSED then
         if cost of new m < old m
            remove old m from CLOSED,
            discard new m,
            make n the parent of old m,
            adjust cost of old m,
            add old m to OPEN
         otherwise
            ignore new m.

8. Reorder the list OPEN, either according
   to some arbitrary scheme or according to
   heuristic merit.

9. Go to step 3.
```

1. ORDERED SEARCH DISCUSSION

Ordered Search maintains only the search tree (and not the state space graph), showing the nodes expanded and the parentage of each (see step 7). Generally, the leaf nodes in the search tree are those on OPEN awaiting expansion, while interior nodes correspond to those on CLOSED. When a node is expanded, some of the children generated will have already been discovered as children of another node. Normally, a newly expanded node will not have been 'seen' before, and is therefore not on OPEN or CLOSED (step 7a). However, if a similar node is found on OPEN or CLOSED, steps 7b and 7c examine which of the two nodes to keep and which to discard on the basis of cheapest cost (we will discuss later precisely what is meant by cost. For now, assume it to refer to the distance between a node and the root of the search tree). If the cost of the new node is greater, the new node is discarded because the path to the new node is longer (might be a loop or simply a similar node just deeper in the tree).

If the new node has a lower cost value, a shortcut has been discovered which needs special handling. If the old node was on OPEN, it is simply discarded and replaced by the new node (step 7b). If the old node was on CLOSED (meaning it has already been expanded and has children), it is removed from CLOSED, the old cost is replaced with the cheaper value, the old node has his parent pointer redirected to the new path, and is then reinserted onto OPEN. The successors of the old node have no way of knowing their parent has been redirected to a new path, and will only discover this when the parent is later reexpanded and they are thus 'recreated' with the new value.

2. ORDERED SEARCH EXAMPLE

Suppose the search process has generated the search tree shown in Figure 5.1. The solid nodes are on CLOSED, and the other nodes are on OPEN at the time the algorithm selects node 1 for expansion; arcs represent parent-child relationship between nodes.

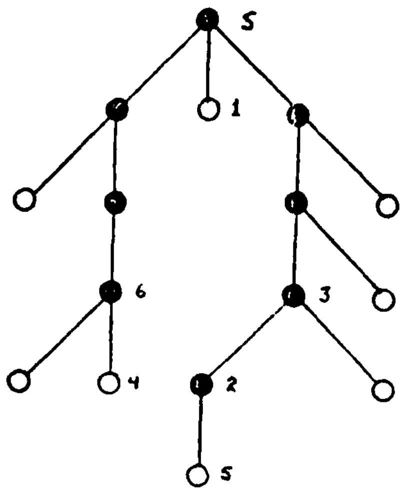


Figure 5.1
Search Tree (Initially)

When node 1 is expanded, its single successor, node 2, is generated (see Figure 5.2). But node 2, with parent node 3 in the search tree, had previously been generated, and node 2 is also on CLOSED with successor node 5. Since the algorithm now discovers a path to node 2 through node 1 that is less costly than the previous path through node 3, the parent of node 2 in the search tree is changed from node 3 to node 1, and node 2 is removed from CLOSED and placed once again on OPEN.

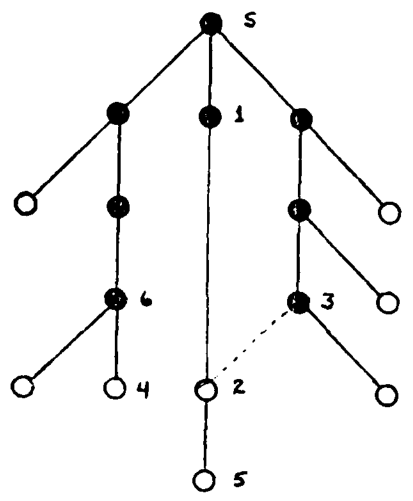


Figure 5.2
Search Tree (Intermediate)

Later, the search algorithm will select node 2 for expansion, generating node 5, and let us suppose node 4 also. Node 4 is already on OPEN with parent node 6, but the cost through node 2 is less, so node 4 has its cost value adjusted and parent altered to node 2. Node 5 is also on OPEN since it was left there last time node 2 was expanded, but its cost is less than before, so its parent is still node 2 but at a lower cost. The adjusted search tree is shown in Figure 5.3.

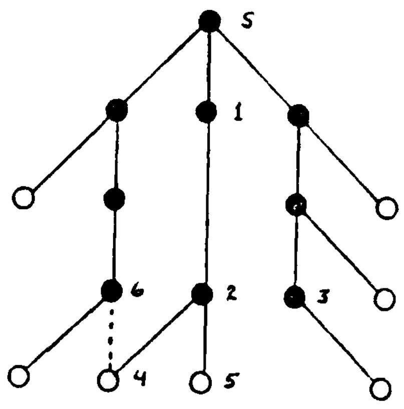


Figure 5.3
Search Tree (Final)

C. GRAPH SEARCH ALGORITHM

Graph Search maintains the search tree to show parentage, but also maintains the structure of the graph itself, showing the multiple parents (or neighborhood). This is needed because when a node is rediscovered, pointers in the search tree may be redirected and F-value changes propogated to the neighborhood in the graph, rather than simply reexpanding all of those nodes. A node, once expanded, is never reconsidered (never put on OPEN again). Maintaining the graph structure adds time and space costs to those already incurred by maintaining the search tree: as nodes are generated, they are not only associated with one parent, they are also linked by arcs to additional parents or neighbors if they are known. Maintaining this graph becomes more expensive as the number of neighbors increases. Here is the Graph Search variation of the A* algorithm provided from Chapter II:

1. Create a new search graph, G, consisting solely of the start node, s. Put s on a list called OPEN.

2. Create a list called CLOSED that is initially empty.

3. If OPEN is empty, exit with failure.

4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this node n.

5. If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G. (Pointers are established in step 7.)

```
6. Expand node n, generating the set M of its
   successors and install them as successors of n in
   G.

7. For each node m in M do one of the following:
   (A) If m not on OPEN or CLOSED then
          establish n as the parent of m,
          add m to OPEN
   (B) If m on OPEN
         if cost of new m < old m
             make n the parent of old m,
             adjust old m to new cost,
             discard new m
         otherwise
             discard new m
   (C)   If m on CLOSED then
          if cost of new m < old m
             make n the parent of old m,
             adjust old m to new cost and
               propogate to old m neighbors
             discard new m
          otherwise
             discard new m.

8. Reorder the list OPEN, either according to some
   arbitrary scheme or according to heuristic merit.

9. Go to step 3.
```

1. GRAPH SEARCH DISCUSSION

The difference between this algorithm and the one presented for Ordered Search is the value propogation performed in step 7c. For each neighbor of m (as per the graph structure), if the neighbor's cost is greater than what it would be going through m, the neighbor's parent is altered pointer to point to m and its cost is adjusted. This change is then propogated to m's neighbors. Propogation is continued until no neighbors are affected by the new value.

2. GRAPH SEARCH EXAMPLE

The following example illustrates the Graph Search process, and is adapted from Nilsson (Ppg 64-66):

Suppose a search process has generated the search graph and search tree shown in Figure 5.4. Since they are superimposed on each other, the dark arrows along certain arcs in this search graph are the pointers that define parents of nodes in the search tree. The solid nodes are on CLOSED, and the other nodes are on OPEN at the time the algorithm selects node 1 for expansion.

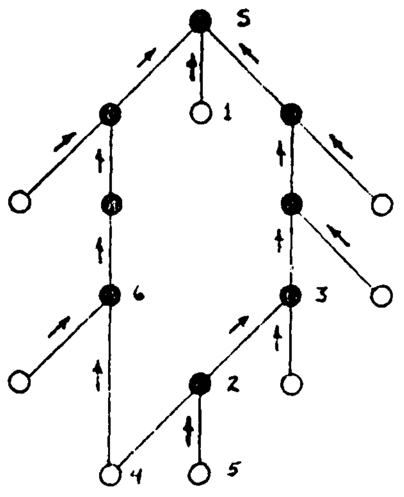


Figure 5.4
Search Graph and Tree (Initially)

When node 1 is expanded, its single successor, node 2, is generated, and installed as a neighbor of node 1. But node 2, with parent node 3 in the search tree, had previously been generated, and node 2 is also on CLOSED with successor nodes 3, 4, and 5. Note, however that node 4's parent in the search tree is node 6, because the shortest path from s to node 4 in the search graph is through node 6. Since the algorithm now discovers a path to node 2 through node 1 that is less costly than the previous path through node 3, the parent of node 2 in the search tree is changed from node 3 to node 1. The costs of the paths to the descendants of node 2 in the search graph (namely, the paths to nodes 3,4 and 5) are recomputed. The costs for nodes 4 and 5 are now also lower than before, with the result that the parent of node 4 is changed from node 6 to node 2. Node 3 is left as it was since the path through node 2 is the same cost as its existing path. The adjusted search tree is defined by the pointers on the arcs of the search graph of Figure 5.5.

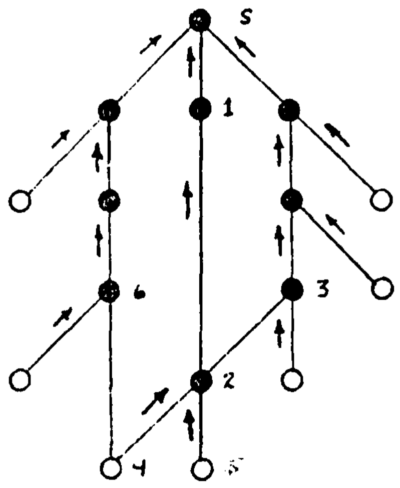


Figure 5.5
Search Graph and Tree (Final)

D. EMPIRICAL COMPARISON RESULTS

We ran both Ordered Search and Graph Search on a common sample of 198 start-goal pairs at varying depths, using 3 different heuristics (the three heuristics are described in the next chapter, but are referred to as K1, K2, and K3) and at 7 different weights, for a total of 4158 problem executions. CPU time for Ordered Search was 15 hours, and for Graph Search was 27 hours. This certainly confirms that the run-time cost of maintaining graph in this setting is very expensive. The space requirements were also greater for Graph Search because additional memory was required to maintain the graph structure.

Graphical comparisons of the results with respect to the total number of nodes expanded and the solution path length found for problems of different depths are included in the pages to follow (Figures 5.6 through 5.17).

Figures 5.6, 5.7, and 5.8 show that the re-expansion effort was so negligible (or nonexistant) for weights less than 0.8 that the curves representing the number of nodes expanded for Ordered Search are superimposed directly over their Graph Search counterparts. At a weight of 0.9, some minor differences between Ordered and Graph search appear (see Figures 5.9, 5.10, and 5.11), and increase dramatically at weight 1.0 (Figures 5.12, 5.13, and 5.14). At weight 0.9, the savings of using Graph Search over Ordered Search averages 5% for K1, 8% for K2, and only 0.9% for K3 (see Table 5.1). At weight 1.0, the savings is much more significant, averaging 53% for K1, 45% for K2, and only 3% for K3 (see Table 5.2). K1 and K2 realized the greatest savings, which is probably because they typically expand more nodes than K3 does so there are chances to encounter duplicates.

The solution paths found by the three heuristics using a weight of 0.9 or less were exactly the same for Ordered Search and Graph Search, as shown by the superimposed nature of the lower curves in Figures 5.15 through 5.17. At weight 1.0, both versions using K3 produced identical path lengths just as they did at 0.9. However, K1 and K2 (Figures 5.15 and 5.16) show that Graph Search produced in most, but not all cases, longer solution paths than Ordered Search did. In K1, the differences were most dramatic and varied. We expected the path lengths to be the same, and provide an explanation for this phenomenon below.

At weight 1.0, the search is purely heuristic in nature and tends to meander down long paths if the heuristic is not accurate. In a graph setting, there are many indirect paths leading eventually to the goal, and all that the A* algorithm guarantees is that of all the paths that are discovered, it will preserve the shortest, even though shorter ones may still exist undiscovered. If an admissible F was used, then A* guarantees that the paths would be minimal (and hence identical in length), but at weight 1.0, F is not admissible. Since the combination of nodes on OPEN in Graph Search is not necessarily the same as in Ordered Search, the two algorithms are likely to expand nodes in an order different from one another. And since many paths can lead to the goal, this accounts for the inconsistent behavior in the path lengths found by the two algorithms. Neither algorithm was consistently better or worse, they just weren't exactly the same in every case either.

Figure 5.6
Graph vs Ordered Search
Heuristics K1, K2, and K3
Weight = 0.2
XMEAN

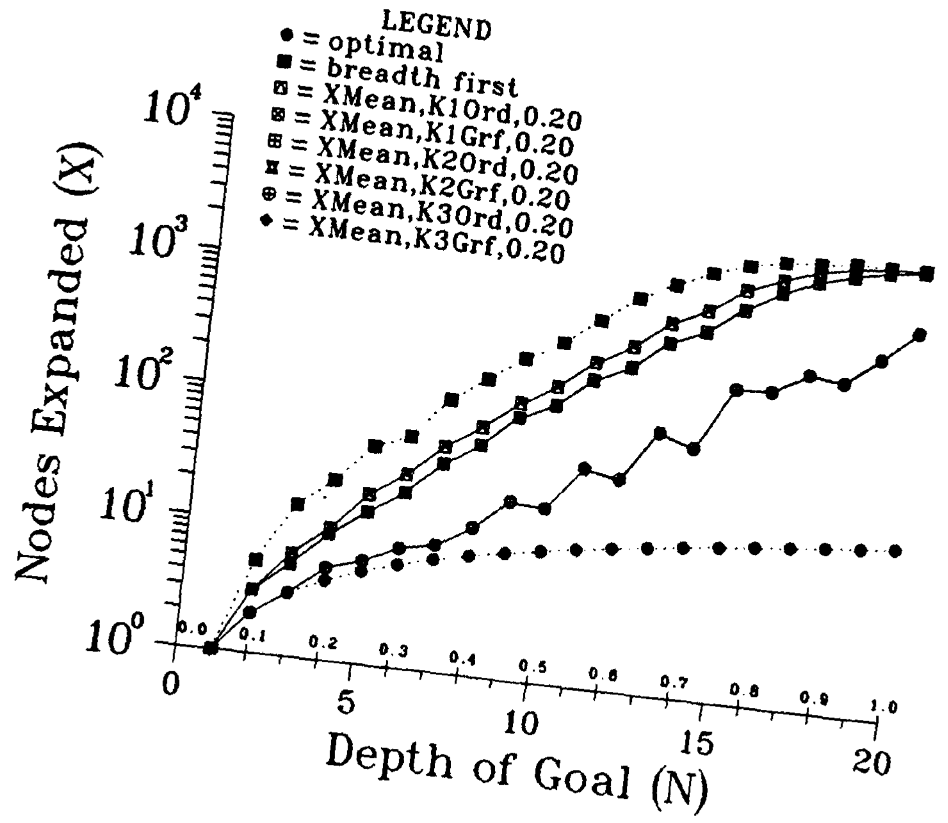


Figure 5.7
Graph vs Ordered Search
Heuristics K1, K2, and K3
Weight = 0.5
XMEAN

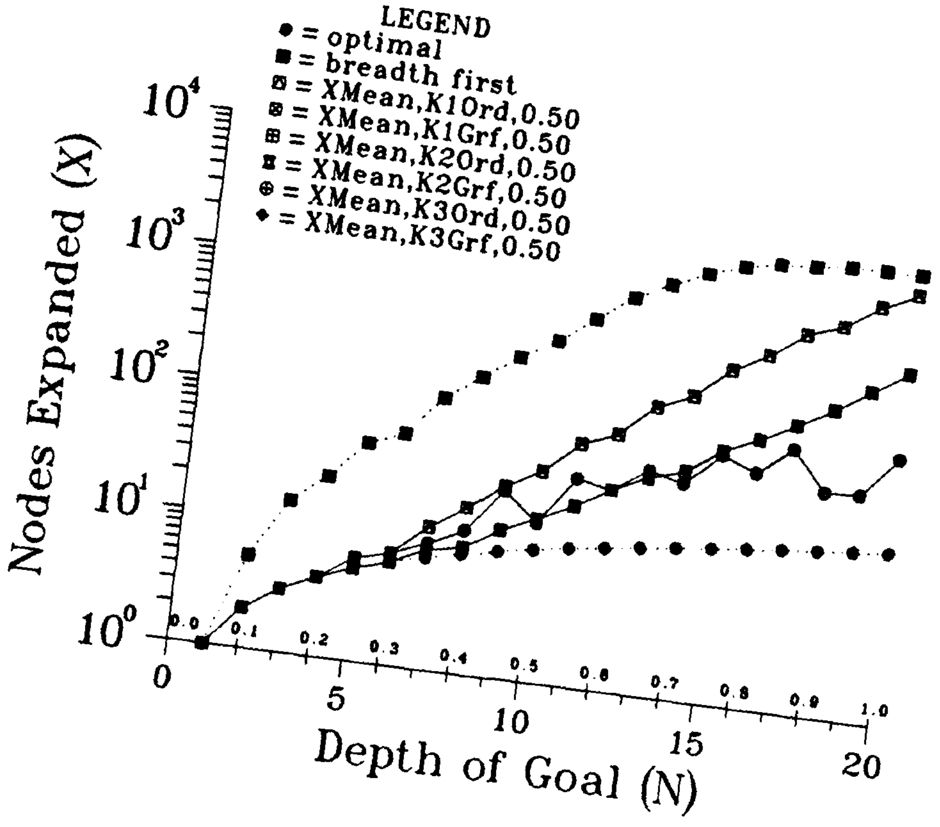


Figure 5.8
Graph vs Ordered Search
Heuristics K1, K2, and K3
Weight = 0.7
XMEAN

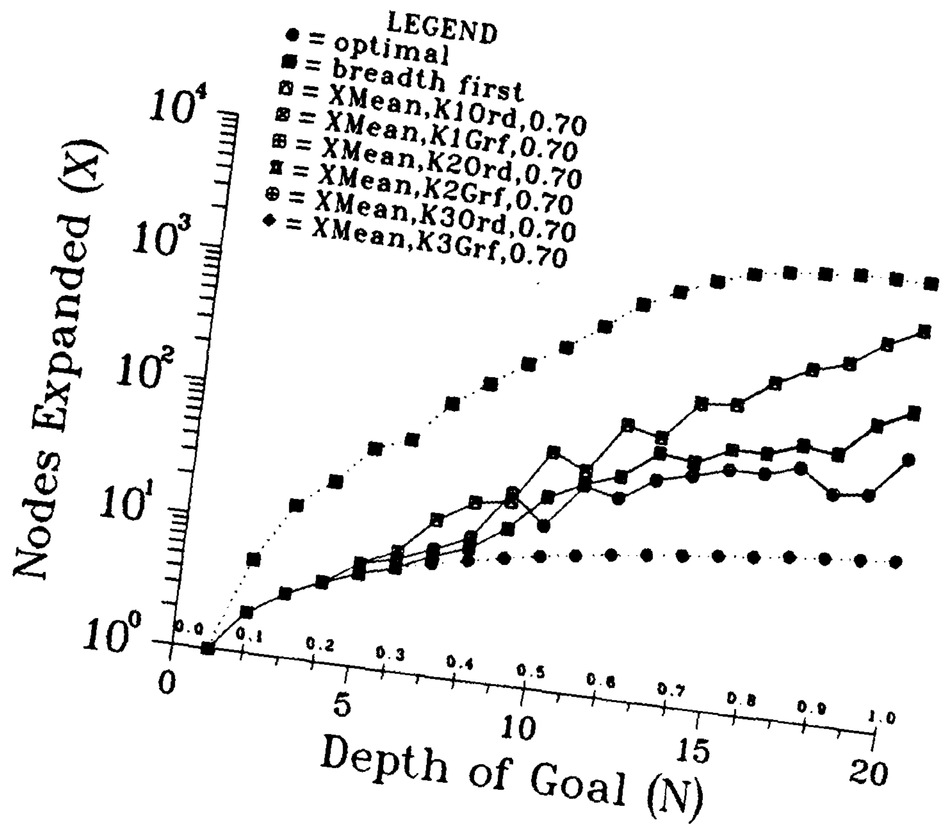


Figure 5.9
Graph vs Ordered Search
Heuristic K1
Weight = 0.9
XMEAN, XMAX

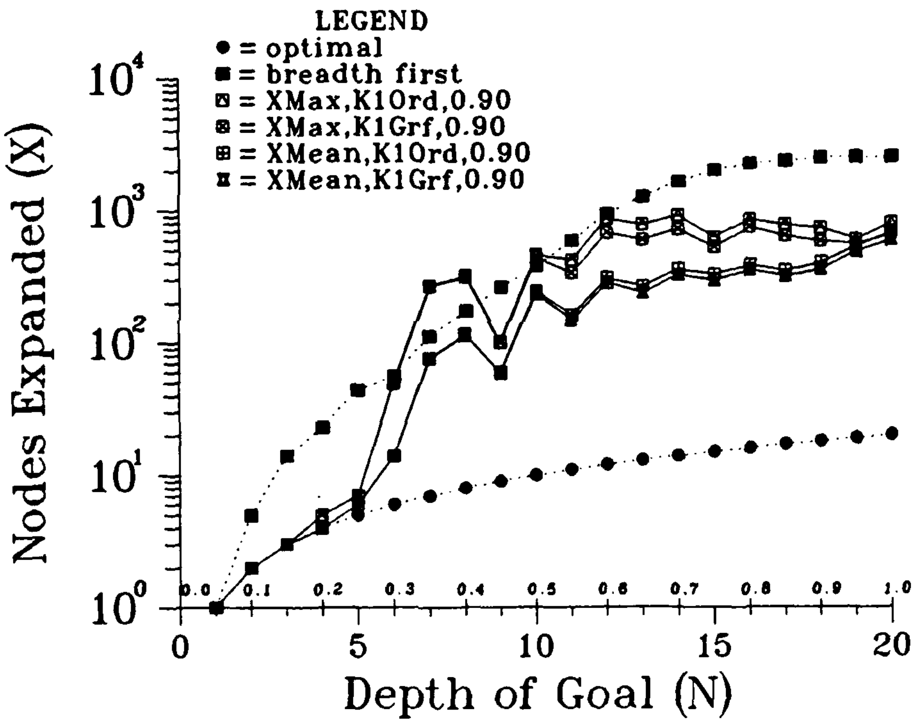


Figure 5.10
Graph vs Ordered Search
Heuristic K2
Weight = 0.9
XMEAN, XMAX

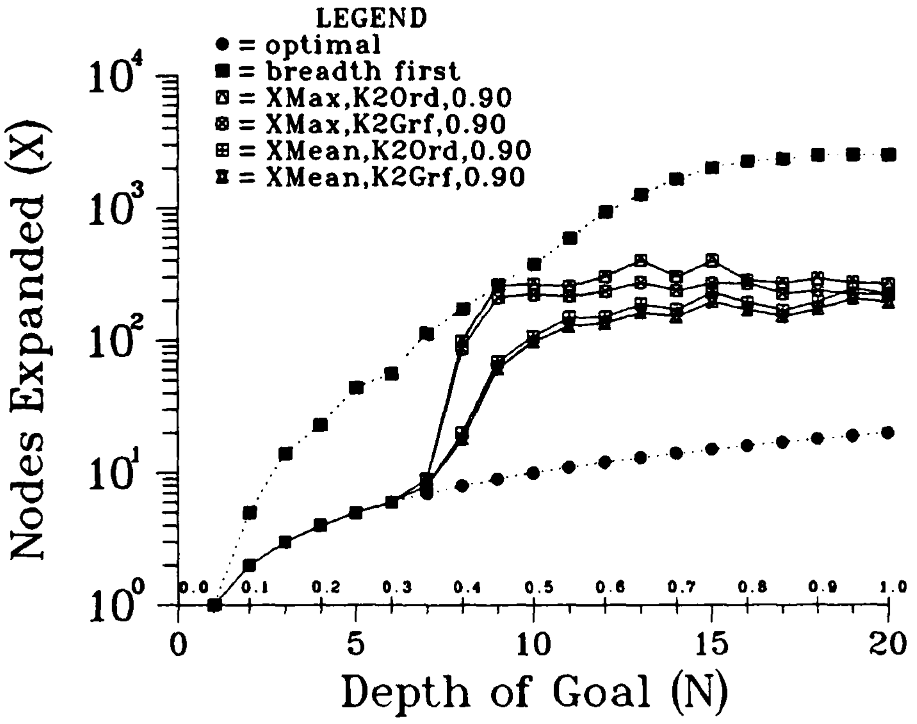


Figure 5.11
Graph vs Ordered Search
Heuristic K3
Weight = 0.9
XMEAN, XMAX

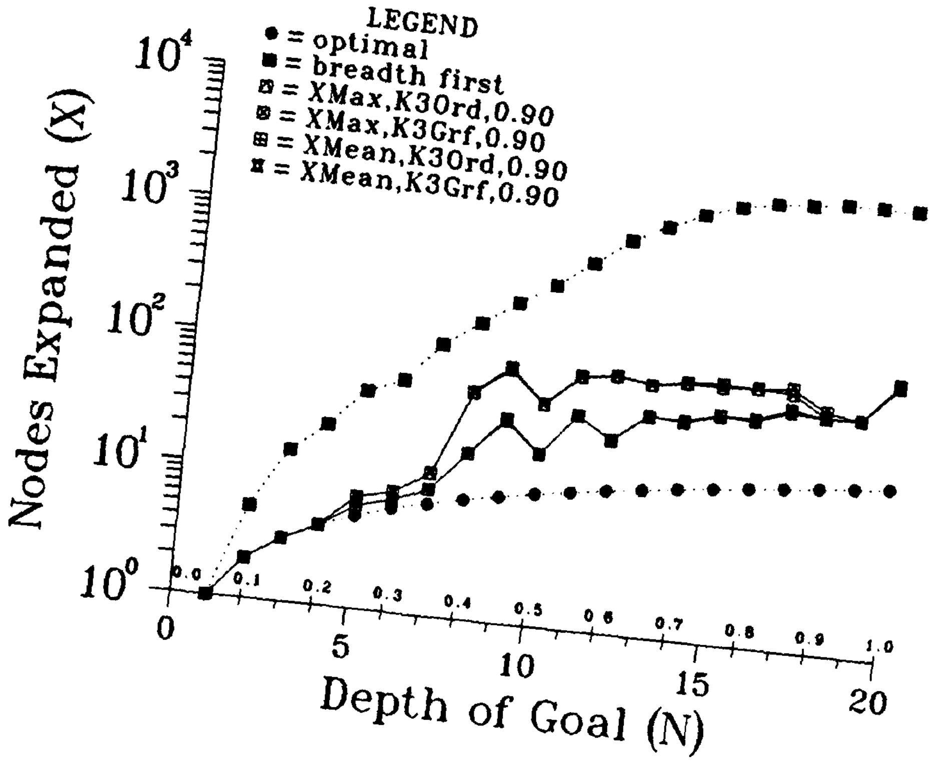


Figure 5.12
Graph vs Ordered Search
Heuristic K1
Weight = 1.0
XMEAN, XMAX

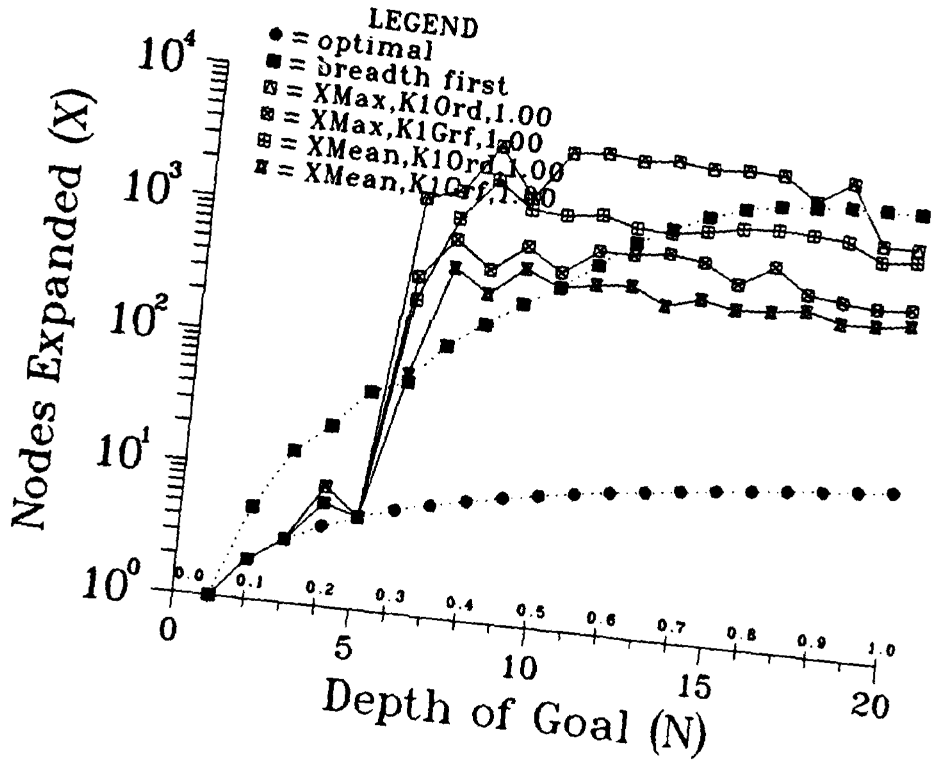


Figure 5.13
Graph vs Ordered Search
Heuristic K2
Weight = 1.0
XMEAN, XMAX

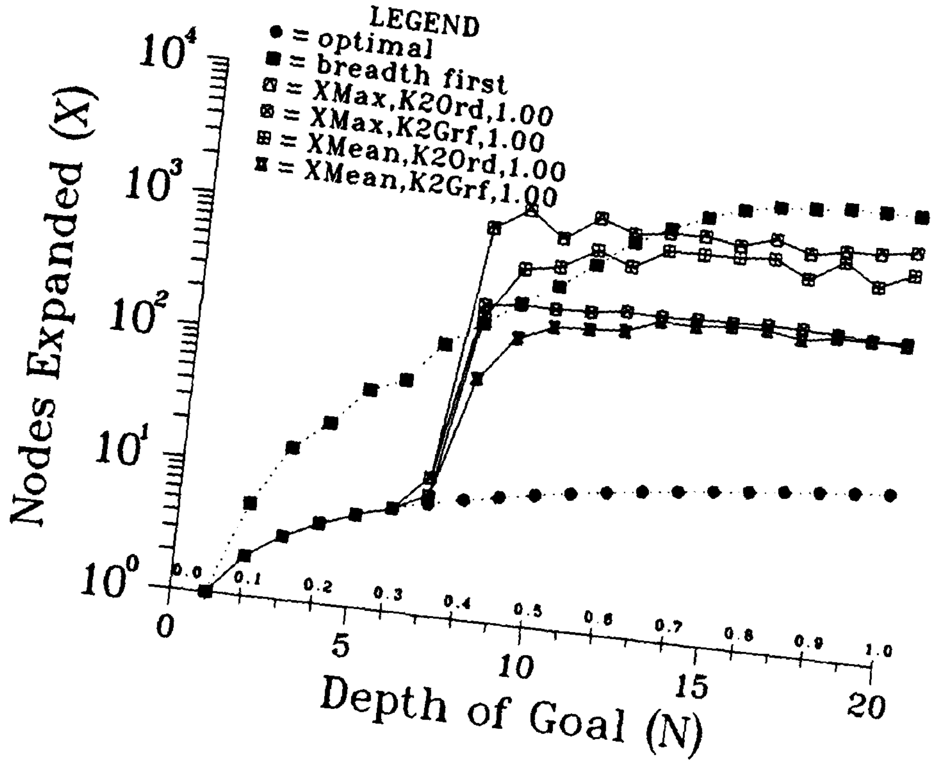


Figure 5.14
Graph vs Ordered Search
Heuristic K3
Weight = 1.0
XMEAN, XMAX

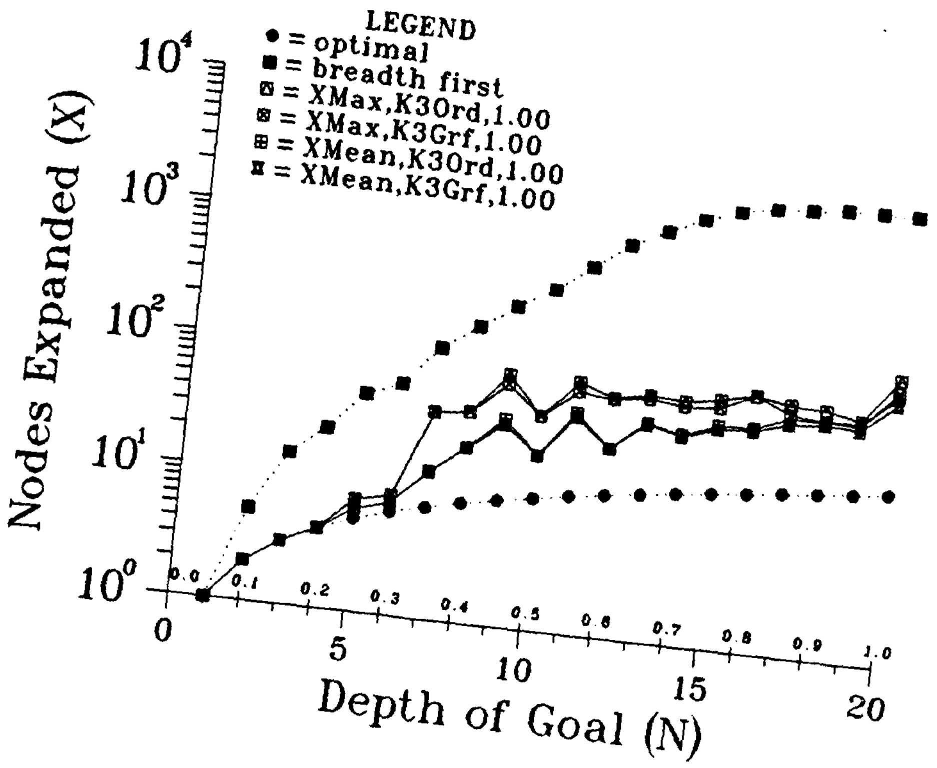


Figure 5.15
Graph vs Ordered Search
Heuristic K1
Weight = 0.9, 1.0
LMEAN

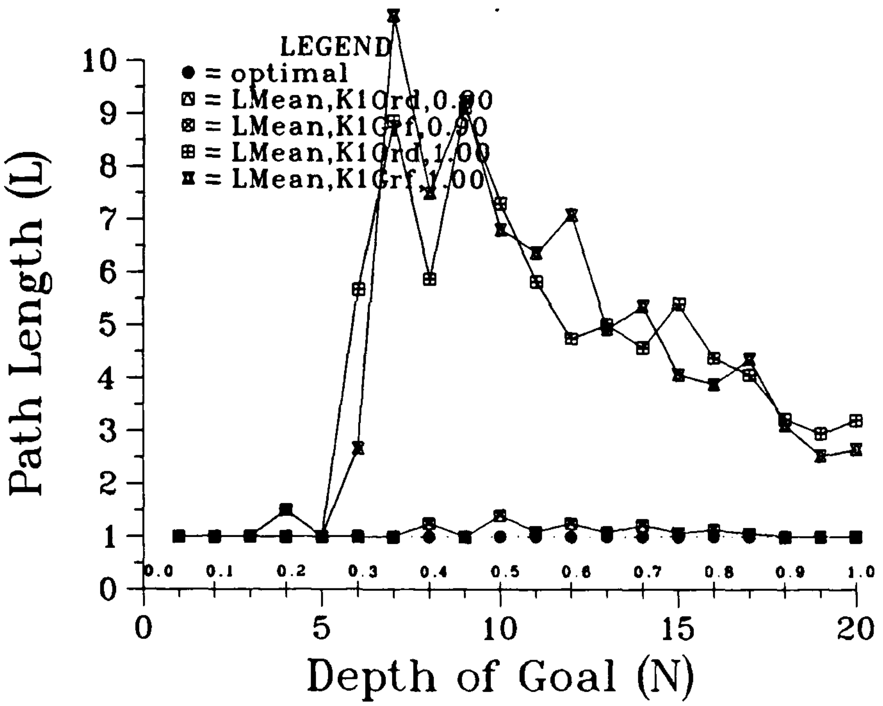


Figure 5.16
Graph vs Ordered Search
Heuristic K2
Weight = 0.9, 1.0
LMEAN

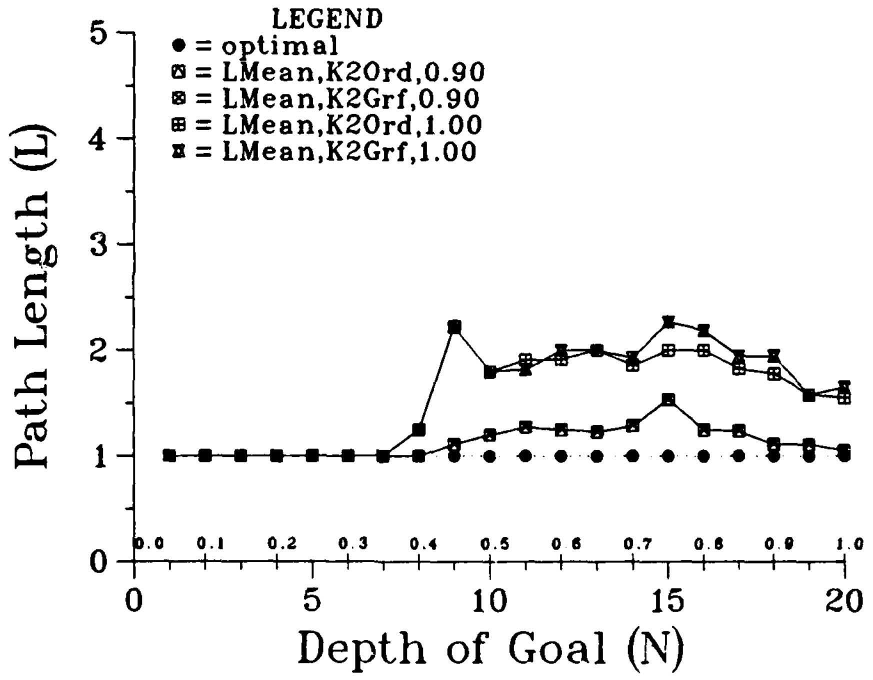


Figure 5.17
Graph vs Ordered Search
Heuristic K3
Weight = 0.9, 1.0
LMEAN

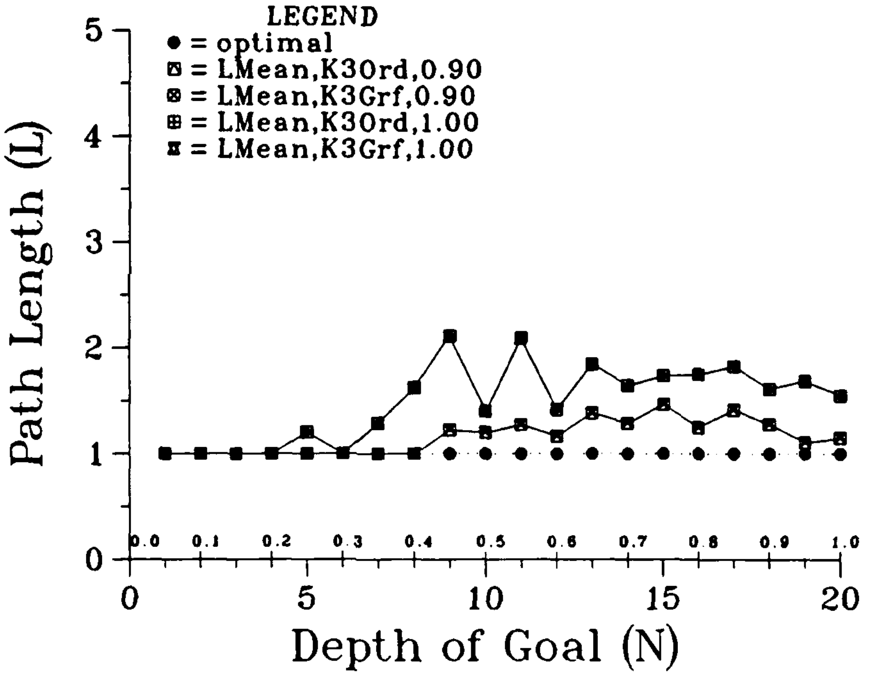


TABLE 5.1
Savings using Graph Search over Ordered Search
Weight = 0.9
(figures expressed in percentages)

	K1		K2		K3	
Level	Mean	Max	Mean	Max	Mean	Max
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	2	0	0	0	0
7	1	1	0	0	0	0
8	2	3	10	12	0	0
9	2	2	12	19	3	4
10	4	4	8	16	0	0
11	7	20	14	16	0	1
12	8	21	10	22	0	1
13	10	23	14	32	0	0
14	9	21	11	21	2	2
15	9	16	16	32	2	5
16	8	12	11	5	4	0
17	9	17	17	10	4	8
18	11	20	12	18	2	9
19	8	7	16	23	0	0
20	13	18	14	17	0	0
	-----	----	----	----	----	----
Ave.	5	9	8	12	0.9	1.5

TABLE 5.2
Savings using Graph Search over Ordered Search
Weight = 1.0
(figures expressed in percentages)

	K1		K2		K3	
Level	Mean	Max	Mean	Max	Mean	Max
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	73	75	0	0	0	0
7	57	57	0	0	0	0
8	86	88	65	74	0	0
9	64	57	70	81	8	15
10	72	87	65	71	0	0
11	70	82	75	80	6	15
12	63	80	67	74	0	2
13	72	80	71	77	2	5
14	69	80	71	76	3	9
15	76	86	70	74	6	13
16	76	80	71	77	4	0
17	73	80	65	73	10	15
18	76	87	73	76	5	17
19	67	66	62	78	11	9
20	67	65	71	79	11	20
	----	----	----	----	----	----
Ave.	53	57	45	50	3	6

E. A* ALGORITHM AMBIGUITIES

1. IMPORTANCE OF TIE-BREAKING POLICY

The reordering of the OPEN list (step 8 of A*) seems rather trivial, but deserves some attention because it can vary performance results. Step 7 generates new nodes or changes some existing nodes' F-values, and step 8 insures that the OPEN list remains ordered since the search algorithm chooses the first node to expand next. Chapter III discussed the impact that ordering this list can have on the resulting search patterns.

The problem is that step 8 isn't explicit about how to order nodes of equal value (referred to as 'breaking ties'). Our method places newest nodes in front of other nodes of the same value, maintaining OPEN in ascending order otherwise. (We also incorporate step 8 into step 7 so that as nodes are generated, we place them on OPEN in ascending order, saving an expensive reorganization of the entire list in a separate step.) This has the effect of encouraging the search deeper along the most recently generated path.

When a node is expanded, its successors are created deterministically and therefore will always occur in the same sequence. This will not change no matter how many times the node is re-expanded. Where two successors have the identical F-value, they will always be inserted onto OPEN in the same order also. Ordered Search, then, will not change this expansion sequence even though a node may be redundantly reexpanded several times. On the other hand, Graph Search only expands a node once and propagates cheaper paths to the applicable successors. This update process involves, for each neighbor of the rediscovered node and their successors, recalculating the new cost, redirecting parent pointers to the new path, removing altered nodes from OPEN and then reinserting them in their new order.

While we know the sequence that successors are generated will never vary, the order of the neighbor list is not predictable nor apparent; nodes are added as they are discovered, yet their order ultimately dictates an order of nodes on OPEN. Suppose two of the descendants end up with the same value. The neighbor updated first will end up behind ones added later because of the tie-breaking policy.

It is difficult to assess the impact of this minor point on the performance of the two algorithms. We feel that even though Graph Search expanded fewer nodes than Ordered Search, Graph Search results could be improved even more by finding a method of avoiding the additional shuffling that the update procedure does to the nodes on the OPEN list. Switching methods from stack (like ours) to queue does not avoid the phenomenon, but merely causes it to manifest itself elsewhere. Certainly there is room for further investigation into this topic.

2. DEFINING 'COST'

An important issue with regard to the Ordered Search and Graph Search algorithms that has not been consistently dealt with in the literature is the cost measure used to determine whether to reexpand or update a rediscovered node. Step 7 of both algorithms base their decision to redirect parent pointers based on a value which until now has generically been referred to as 'cost'.

Nodes are ordered on OPEN on the basis of their F value, which is comprised of a G component (distance from root of search tree) and an H component (estimate of distance remaining to the goal). The discriminator used in step 7 could be either the G component or the F value. Since any rediscovered node will always calculate to the same H, the only way to tell if the path is shorter is by examining the G component. In the unweighted version of A*, either F or G could be used with no effect on the results, because if G changes, F also changes. In the Weighted A* algorithm (which we used), this is still the case for all weights less than 1.0. In this one special case, F is based purely on the heuristic component (H), and G is given no weight at all. Therefore, when F is used as the discriminator, each rediscovered node will always have the same F value at weight 1.0, and will be automatically discarded because it's F-value doesn't involve the G component to inform the program that it is on a shorter path. This is not the case, however, if G is used as the discriminator in step 7.

Empirically, this means that at W=1.0, the Weighted A* algorithm using Ordered Search with F as the discriminator should expand fewer nodes than the same version using G, but ought to find shorter paths. Since Graph Search never reexpands nodes, the only difference should be that Graph Search using F would find longer paths than using G, but nodes expanded should be the same.

a. RESULTS OF USING F OR G FOR COST DISCRIMINATOR

The results for the Ordered/Graph Search comparison presented so far were generated using G as the discriminator in step 7, not only because it returned the most accurate results, but also because it preserved the nature of the A* algorithm with no special cases. However, the next chapter compares the 6-Puzzle to Gaschnig's 8-Puzzle; since he used Ordered Search with F as the discriminator, in order to provide a direct comparison, we used the same (Ordered Search using F). Thus, we can also compare the results of using F and G as discriminators within the Ordered Search algorithm.

Figures 5.18 through 5.26 present the results generated using Ordered Search at various weights using F versus G as the discriminator (the lower graph). Figures 5.18 through 5.20 show that there is no difference between the two discriminators using K1, K2, and K3 at weights 0.2, 0.5, and 0.7 since the performance curves for the version using F are superimposed on top of their G version counterparts. However, at weight 1.0, significant differences are observed, with the G version expanding much more nodes than the F version (Figures 5.21, 5.22, and 5.23). The observed differences are greatest for K1 and very small for K3.

The lower curves in Figures 5.24 through 5.26 show that the path length discovered at weight 0.9 is identical between the G version and the F version of Ordered Search. The results observed at weights lower than 0.9 were also identical for both versions, but their graphs are not included to conserve space. Notice that the path lengths at weight 1.0 were different. The path length for K1 was much higher for the G version (Figure 5.24), but not consistently better at every N. The path lengths for K2 (Figure 5.25) were only slightly higher for the G version, but again, not consistently for every N. The path lengths for K3 (Figure 5.26) reverse the trend, and show that the F version produced longer solution paths for every N.

It was expected that the two versions would be identical in the lengths of the solution paths found, so the variations observed above were somewhat surprising. The explanation for this behavior is the same as the reason given for the differences in path length between Ordered Search and Graph Search in Chapter V section D. The OPEN list of each variation does not contain the same number or combination of nodes, and so the resulting search patterns will vary slightly with the possibility that different paths to the goal are discovered. The A* algorithm only guarantees that of the paths discovered, the shortest will be reported.

Figure 5.18
Ordered Search Discriminator Comparison
Heuristics K1, K2, K3
Weight = 0.2
XMEAN

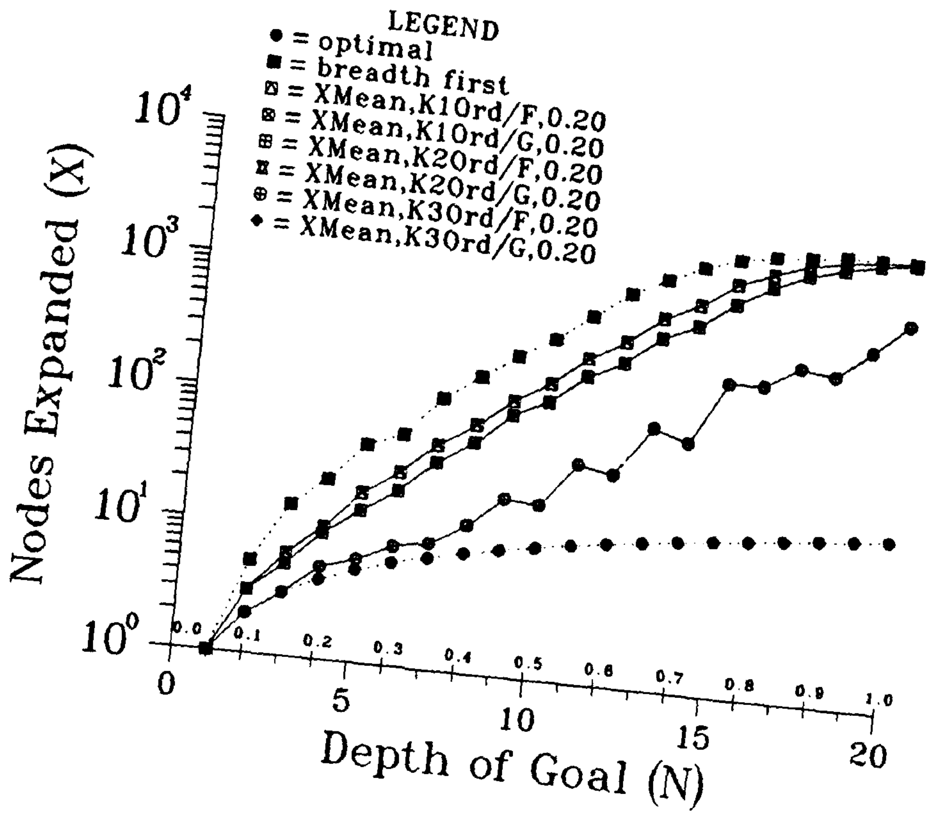


Figure 5.19
Ordered Search Discriminator Comparison
Heuristics K1, K2, K3
Weight = 0.5
XMEAN

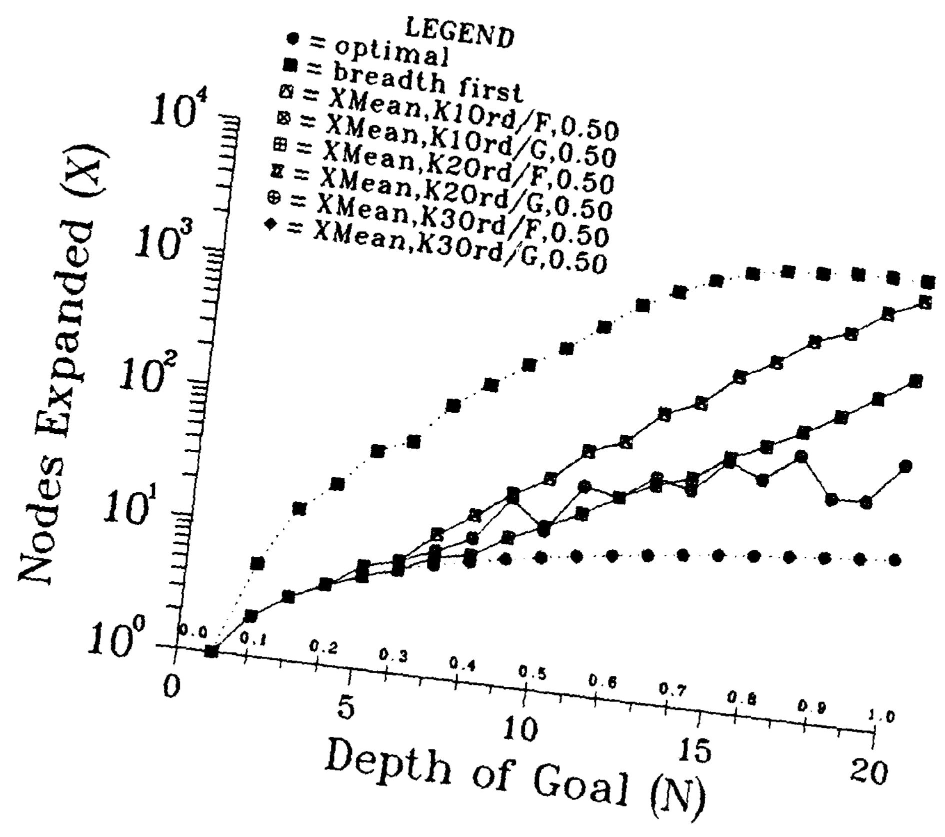


Figure 5.20
Ordered Search Discriminator Comparison
Heuristics K1, K2, K3
Weight = 0.7
XMEAN

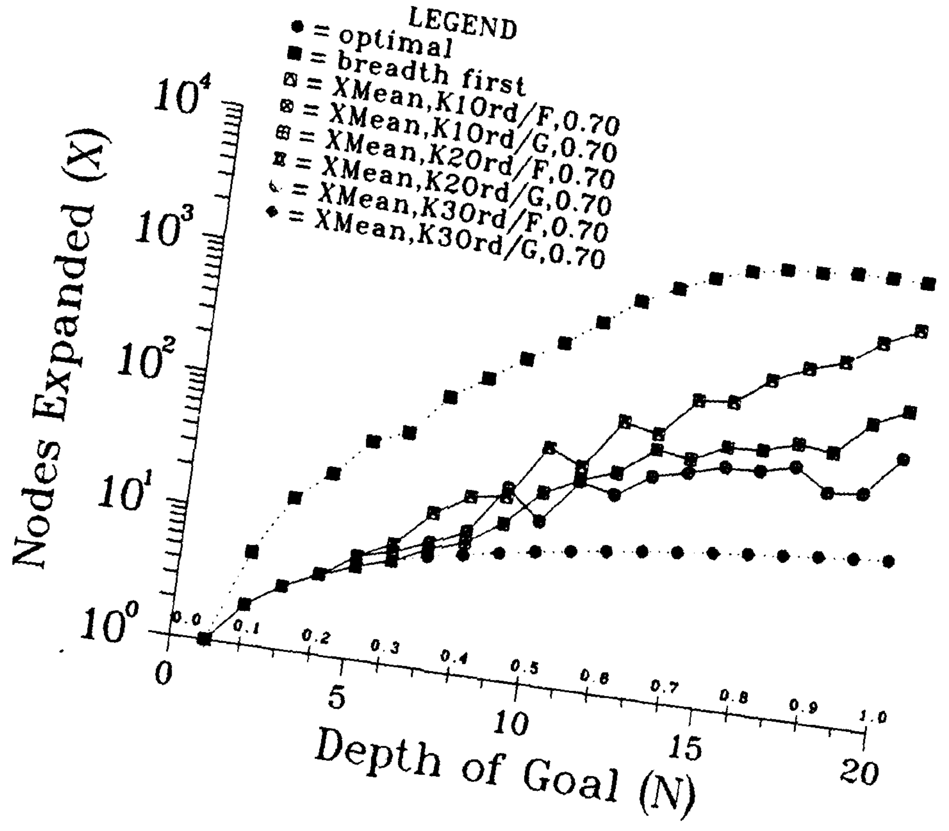
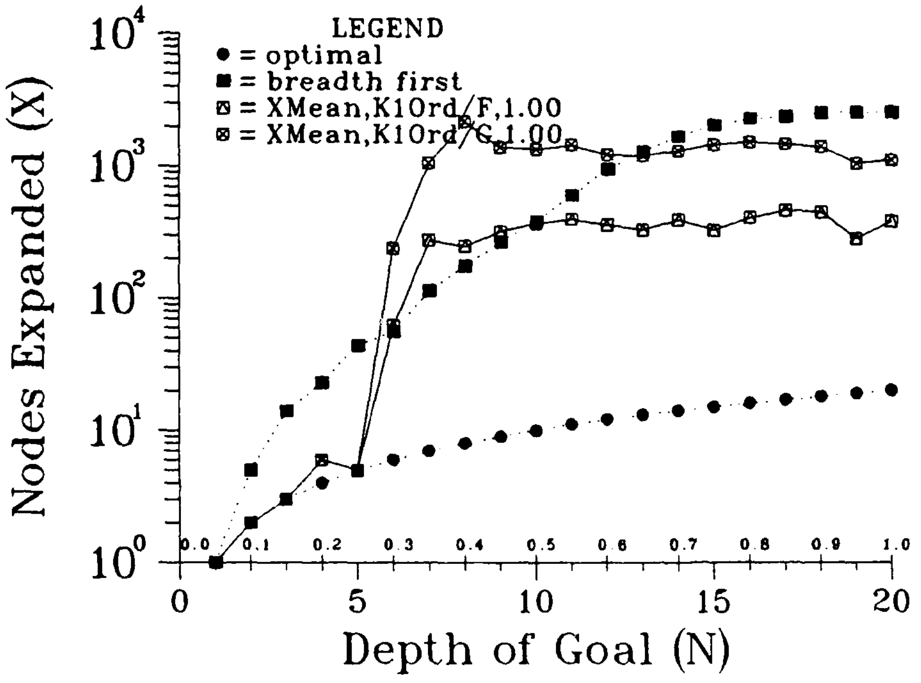


Figure 5.21
Ordered Search Discriminator Comparison
Heuristic K1
Weight = 1.0
XMEAN



AD-A173 096 AN EMPIRICAL STUDY IN THE SIMULATION OF HEURISTIC ERROR BEHAVIOR(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH S R HANSEN 1986 AFIT/CI/NR-86-184T 2/4

UNCLASSIFIED F/G 9/2 NL

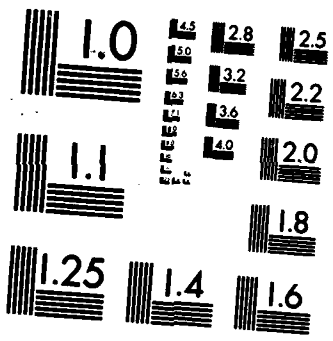

1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

Figure 5.22
Ordered Search Discriminator Comparison
Heuristic K2
Weight = 1.0
XMEAN

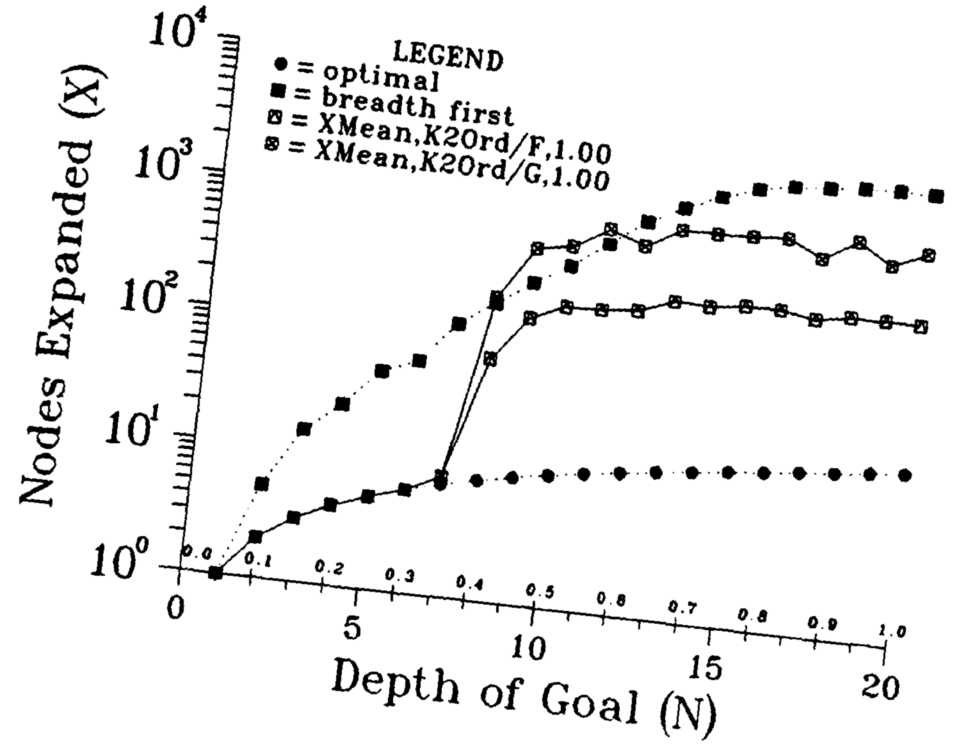


Figure 5.23
Ordered Search Discriminator Comparison
Heuristic K3
Weight = 1.0
XMEAN

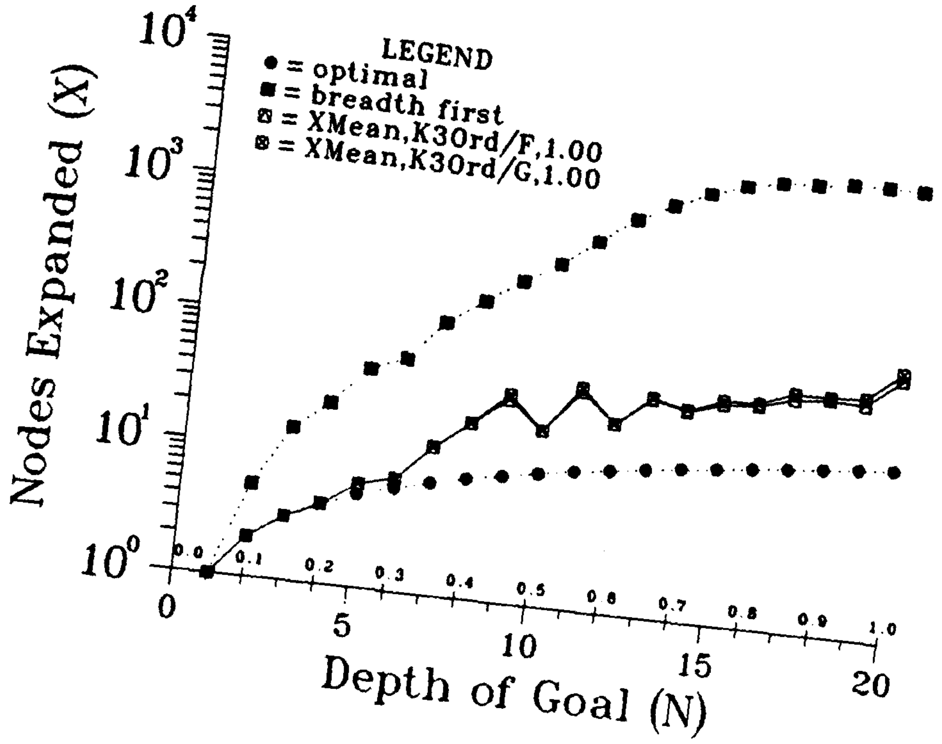


Figure 5.24
Ordered Search Discriminator Comparison
Heuristic K1
Weight = 0.9, 1.0
LMEAN

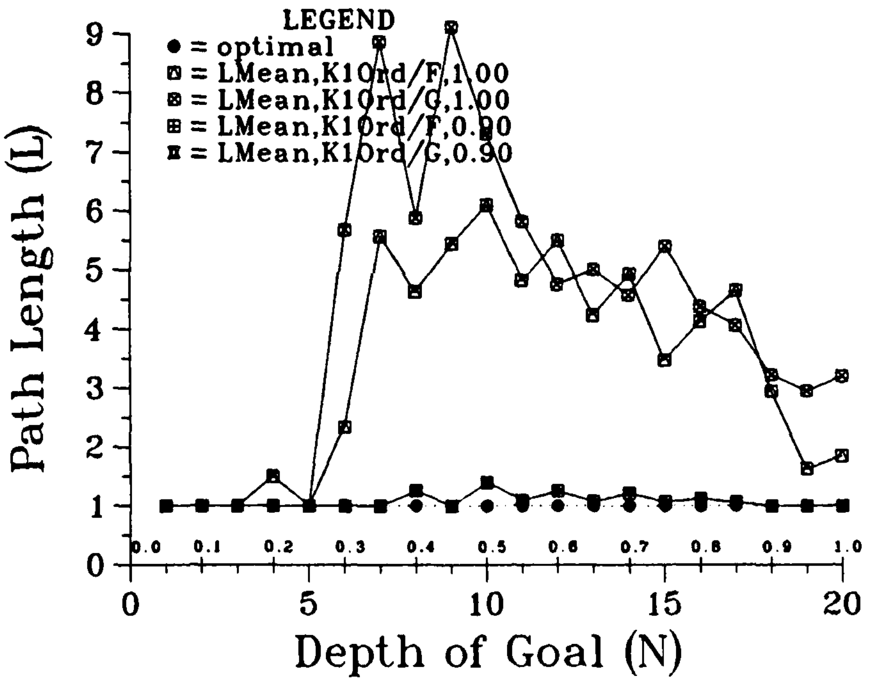


Figure 5.25
Ordered Search Discriminator Comparison
Heuristic K2
Weight ↔ 0.9, 1.0
LMEAN

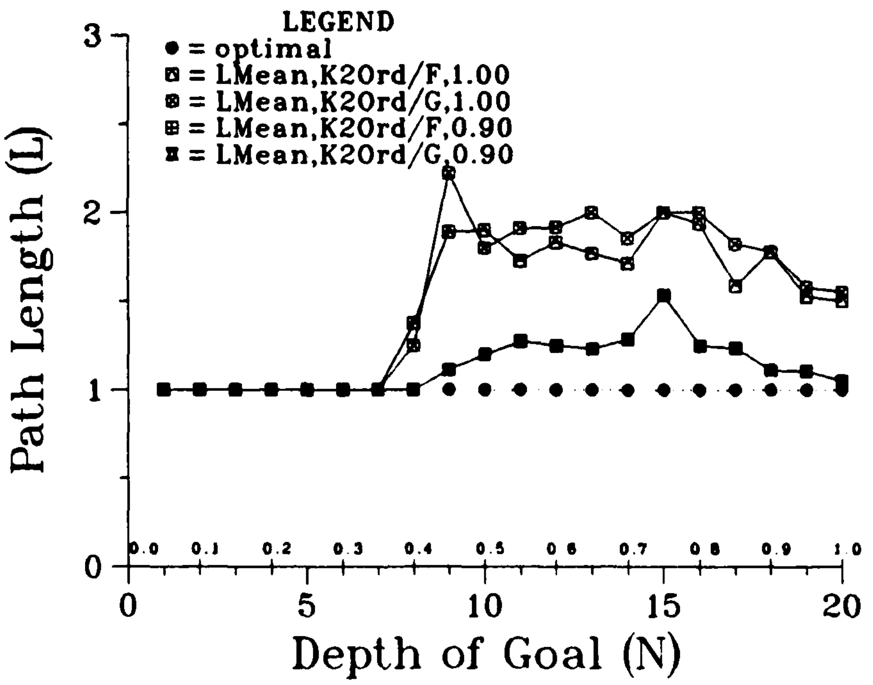
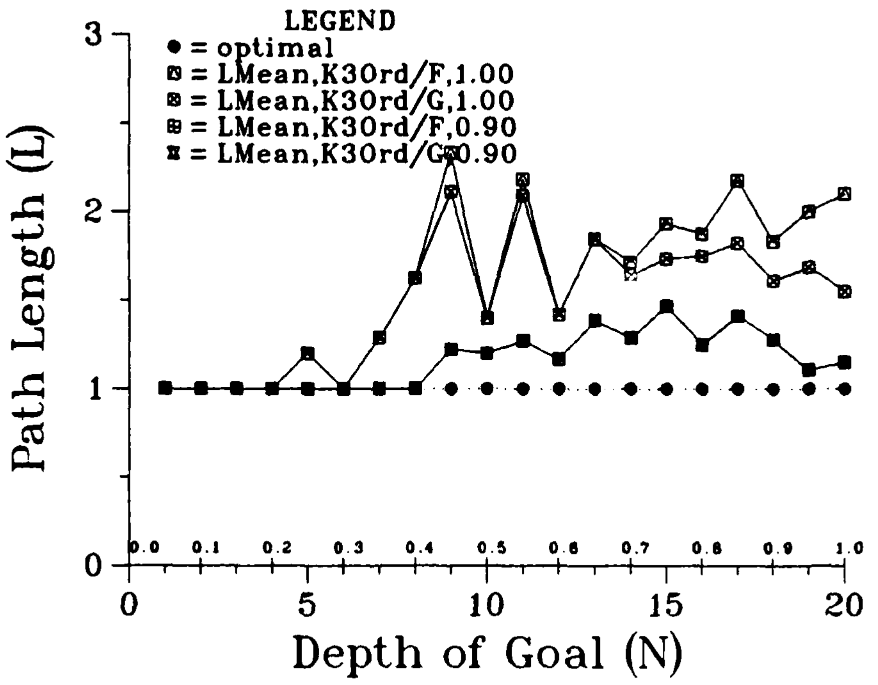


Figure 5.26
Ordered Search Discriminator Comparison
Heuristic K3
Weight = 0.9, 1.0
LMEAN



F. CONCLUSIONS

It is common in the literature to take the number of nodes expanded as the measure of time complexity for "best-first" search algorithm studies (such as ours). In this context, Graph Search is clearly superior. However, our studies show that in general, the number of nodes expanded in and of itself is not an adequate measure of run-time complexity for best-first search, and that CPU time must also be considered.

What are our recommendations? It depends. In this domain, Graph Search took twice as much time as Ordered Search, but only showed a savings in terms of nodes expanded at weights 0.9 and 1.0. The mean savings at 0.9 was less than 10%, while the savings at a weight of 1.0 was much higher, averaging around 50% for K1 and K2. At lower weights, this suggests that the cost of using Graph Search outweighs the benefits, and Ordered search appears to be the better choice. At higher weights, however, one must place priority on either optimizing execution time (favoring Ordered Search), or optimizing the number of nodes expanded (favoring Graph Search).

Not all domains are similar to the 6-Puzzle's, and there is a tradeoff to be considered. More complex graph structures favor Ordered Search because of the added expense (in computer time and memory requirements) of maintaining the graph structure. If the cost of generating a node is quite high (suppose the rules for successor creation are very elaborate and non-trivial, or that the cost of obtaining dynamic storage from the system is high` , then Graph Search would be favored.

On the issue of choosing F or G as the discriminator for redirecting parent pointers, our results show a significant savings in nodes expanded using Ordered Search at a weight of 1.0 with F as the discriminator. (Graph Search nicely sidesteps the issue of the discriminator's effect on the number of nodes expanded.) However, a judgement must be made on either using G and preserving the nature of the A* algorithm at the expense of much poorer performance at weight 1.0, or opting for better performance by using F as the discriminator and introducing a special case into the A* search algorithm.

VI. 6-PUZZLE/8-PUZZLE COMPARISON

In a previous chapter, we indicated that one configuration of links in the 6-Puzzle family appeared strikingly similar to the 8-Puzzle. In this chapter, we compare the two puzzles to each other by empirically comparing the performance of A* using the same heuristics on both. Besides simply comparing the 6-Puzzle to another domain, establishing this similarity is important because then further experiments performed on the 6-Puzzle could yield results which may be considered more general, and also valid in the 8-Puzzle domain. This is especially attractive because experimentation on the 6-Puzzle is much more cost-effective.

Gaschnig (1979) did extensive research on the 8-Puzzle using three heuristic functions and at a variety of weights. In effect, he held the domain fixed and varied the heuristics; we did the same using the 6-Puzzle. We moved his three heuristics into the 6-Puzzle domain and conducted the same series of executions at a variety of weights just as Gaschnig did. In effect, we not only held the domain fixed and varied the heuristics, but by comparing our results with the his, we are also able to compare the result of holding the heuristics fixed and varying the domain!

A. 8-PUZZLE HEURISTICS

A brief discussion of heuristics and the F function was given in Chapter II. Their purpose is to guide the search process by intelligently ordering nodes on OPEN so that the most promising nodes are expanded first, hopefully leading directly to the goal without detours along the way. Basically, we used heurstics that estimated the number of moves remaining to reach the goal by examining certain aspects of the current puzzle's configuration. These are the definitions Gaschnig gave for the three heuristics he used in his empirical studies with the 8-Puzzle:

"K1 = number of tiles that occupy a board location in s different from the location occupied by that tile in the goal node.

K2 = the sum, over all 8 tiles in s, of the minimum number of moves required to move the tile from its location in s to its desired location in the goal node, assuming that no other tiles were blocking the way.

K3 = K2 + 3 * SEQ(s)
where SEQ(s) counts 0 if the non-central squares in s match those in goal up to [one] rotation about the board perimeter, and counts two for each tile not followed (in clockwise order) by the same tile as in the goal node."

K1 is simply a count of the number of tiles 'out of place' in the puzzle. The number of moves remaining to the goal will never exceed the number of tiles out of place, so this heuristic always underestimates the distance remaining. It also has an upper bound in that no more than 8 tiles can be out of place because that is the number of tiles in the puzzle.

K2 (also known as Manhattan Distance, or the "city-block" distance) counts the number of positions each tile is out of place. This represents the number of moves it would take each out-of place tile to get into its goal position if tiles could move over each other (which they cannot do in real life). This heuristic provides a more realistic estimate than K1 does and, like K1, is an under-estimater, since tiles blocking the path must be dealt with.

K3 (also called the Enhanced Manhattan Distance) uses a combination of K2 and a measure called SEQ. The purpose of SEQ is to assess the relative placement of perimeter tiles with each other, assigning a numeric penalty for tiles not followed by the proper "next" tile, and in essence, giving the estimate of the number of moves required to swap the order of two inverted tiles. Note that this heuristic can overestimate the actual distance to the goal, distinguishing it from K1 and K2.

B. MOVING THE HEURISTICS TO THE 6-PUZZLE

Both K1 and K2 were easily encoded for the 6-Puzzle. The definition given for K3 was somewhat ambiguous, however. The ambiguity lay in how to deal with a blank in the perimeter: if tile 7 is followed by a blank and is supposed to be followed by tile 8, should this situation count 2 or not?

To ensure that our encoding of the three heuristics was correct, especially in the case of K3, we ran them against a variety of start/goal puzzle states, and just collected the heuristic estimates versus the actual distance to the goal for each node expanded enroute to the goal. This allowed us to compare our heuristics' estimates to Gaschnig's since he used the same technique to collect values from his, and reported these values in his dissertation (see Figures 6.19a, 6.20a, and 6.21a).

We found amazing agreement with Gaschnig for K1 and K2 (see Figures 6.19b and 6.20b). Our K3 was grossly overestimating, however. So we modified K3 to ignore any bead comparisons involving a blank position but to count 2 for every bead not immediately followed (clockwise) by the bead supposed to be there as dictated by the goal state.

After running this version of K3, the new estimates compared more favorably to Gaschnig's, but were still somewhat overstated (see Figure 6.21a and b). We feel that some of this is due to the factor 3 by which SEQ is multiplied. This factor, originally obtained by empirical study, must capture something that is unique to the 8-Puzzle and should probably be adjusted for the 6-Puzzle.

We defend our use of the second implementation of Gaschnig's K3 by giving an example where the result of counting the blank is not consistent with the results

Gaschnig reported:

Start

1	2	3
	8	4
7	6	5

Goal

1	2	3
8		4
7	6	5

The example shows the goal one move away from the start state (see tile 8). If blanks are counted in the SEQ calculation, tile 7 is not followed clockwise by tile 8 and therefore would count 2. K3 would return a 7 in this case since the Manhattan Distance is 1 (tile 8 is one move from its goal position and all the others are in their proper positions), and we multiply SEQ by 3. Figure 6.21a shows that Gaschnig's K3 always returned the value 1 when the distance to the goal was 1. If, on the other hand, the blank is not counted, the comparison of tile 7 with the blank is ignored, SEQ is 0 (finding all remaining perimeter nodes in their relative orders), and K3 returns only the Manhattan Distance (K2), which is 1. This agrees with what Gaschnig reported and is the form in which we employed K3. We feel that to achieve exact agreement between our K3 and the 8-Puzzle's K3, some adjustment of the factor 3 by which SEQ is multiplied is in order, although we did not pursue this idea in this thesis.

C. EMPIRICAL RESULTS

We anticipated that the results of using K1 and K2 would match closely with Gaschnig's, but that K3 may not be as good. Gaschnig used the Ordered Search variation of the Weighted A* algorithm, with F as the discriminator for node reexpansion (described in the previous chapter). To provide a fair comparison of results between our work and his, we implemented the same variations.

The following pages contain graphs (Figures 6.1 through 6.21) representing the execution of our heuristics upon a sample of 198 start-goal pairs, using weights of 0.2, 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0. Each figure shows both the 6-Puzzle (lower graph) and the 8-Puzzle (upper graph) results to simplify their comparison as much as possible. We attempted to duplicate his format as closely as possible. However, there will be some slight differences in axis scalings and symbols. We will not attempt to summarize each of the graphs, but invite the reader to compare for himself the striking similarity between the two domains, and refer him to Gaschnig's dissertation for a thorough, eloquent description of their individual meanings. While 'eyeing' in graphs is not a mathematically precise method of comparison, it does suffice in this case. A summary and conclusion follows the graphs.

(Please refer to Appendix E for a description of the terms used in the graphs.)

Figure 6.1
XMIN, XMEAN, XMAX vs N
Heuristic K1
W = 0.5

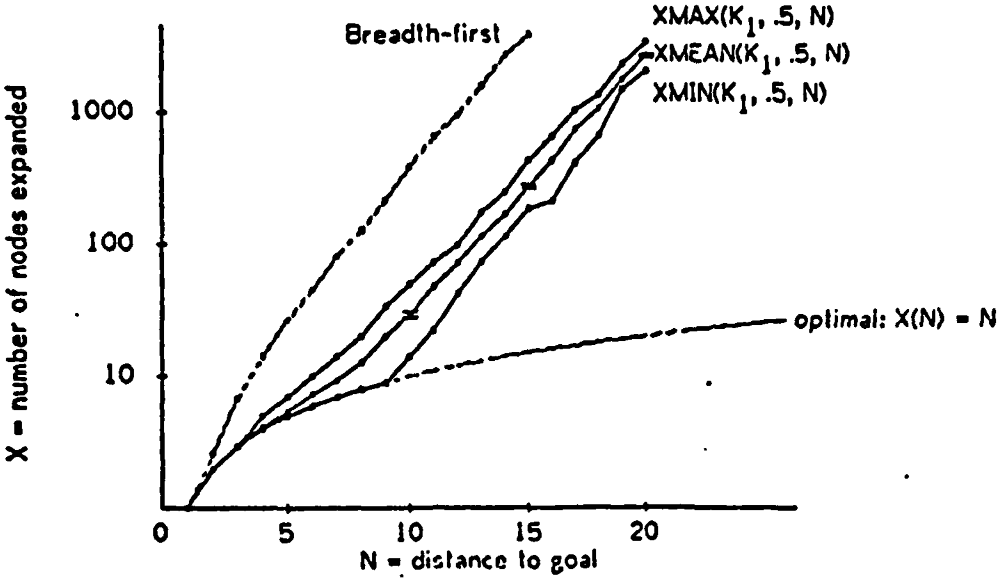


Figure 6.1a 8-Puzzle

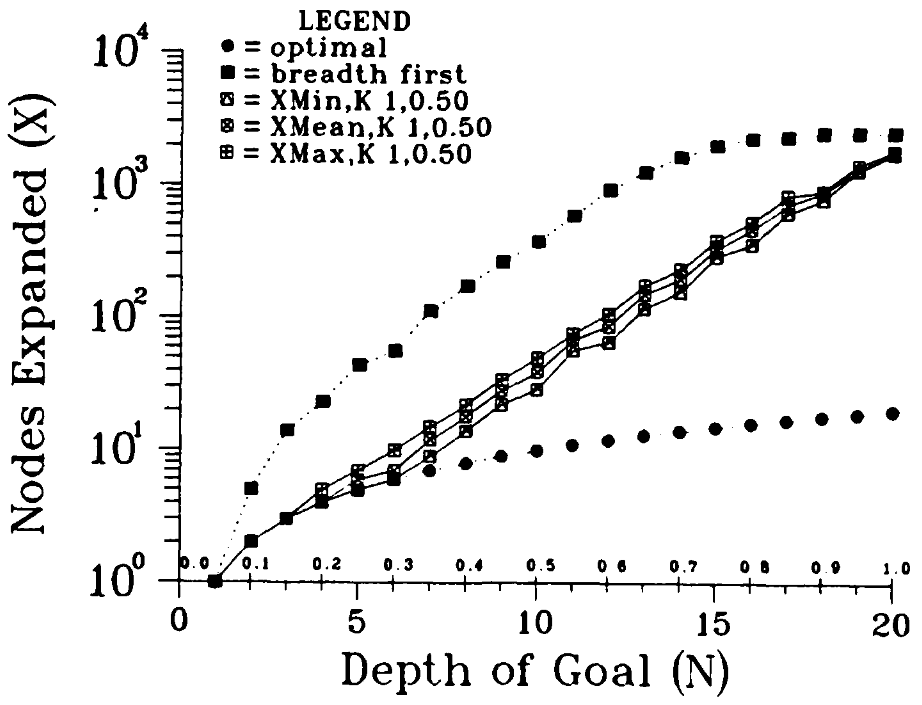


Figure 6.1b 6-Puzzle

Figure 6.2
XMIN, XMEAN, XMAX vs N
Heuristic K2
W = 0.5

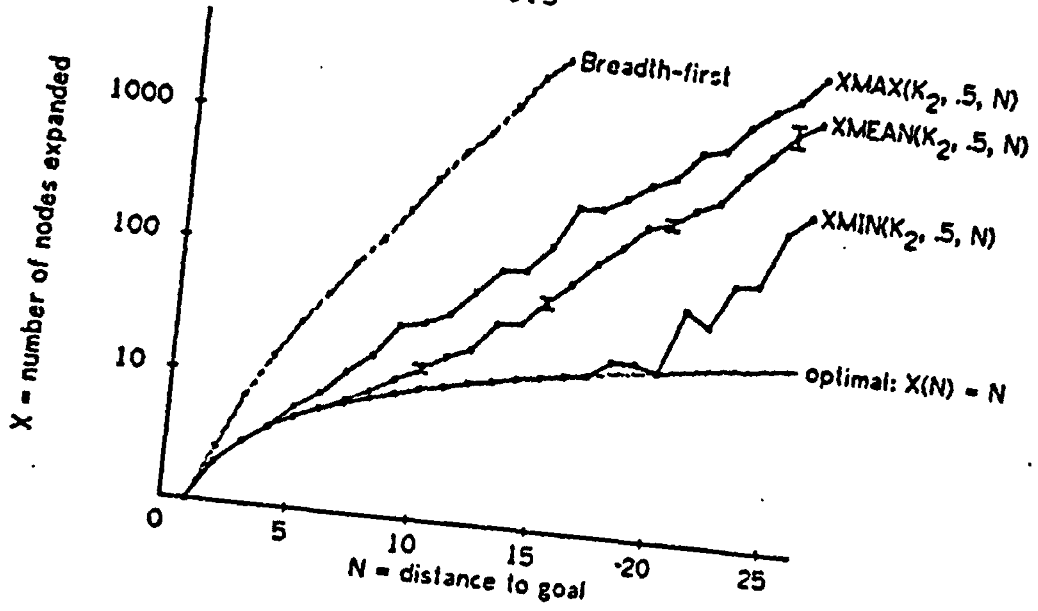


Figure 6.2a 8-Puzzle

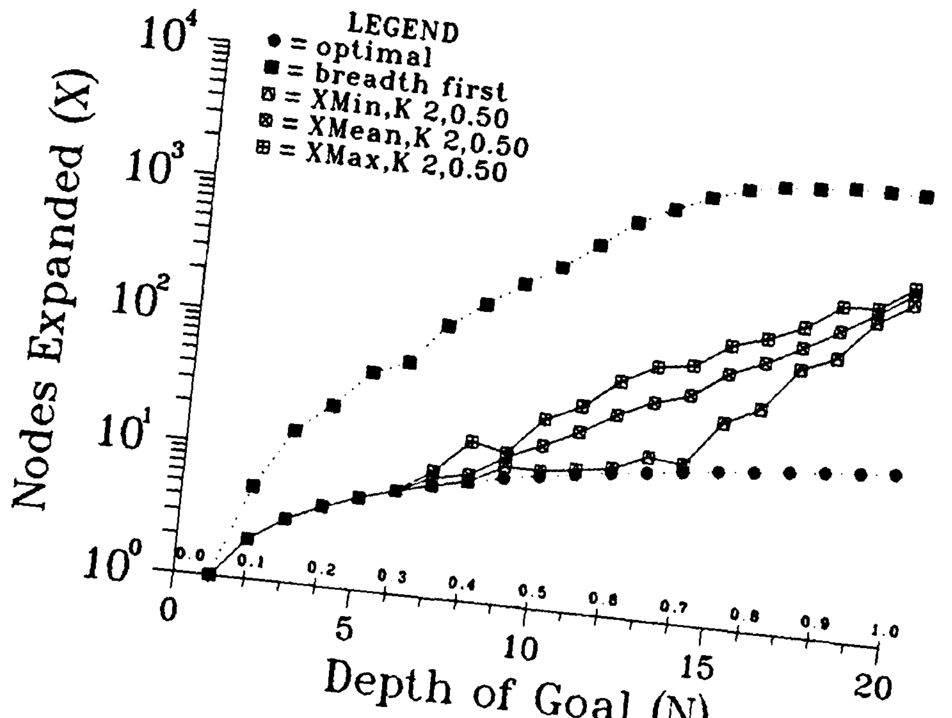


Figure 6.2b 6-Puzzle

Figure 6.3
XMIN, XMEAN, XMAX vs N
Heuristic K3
W = 0.5

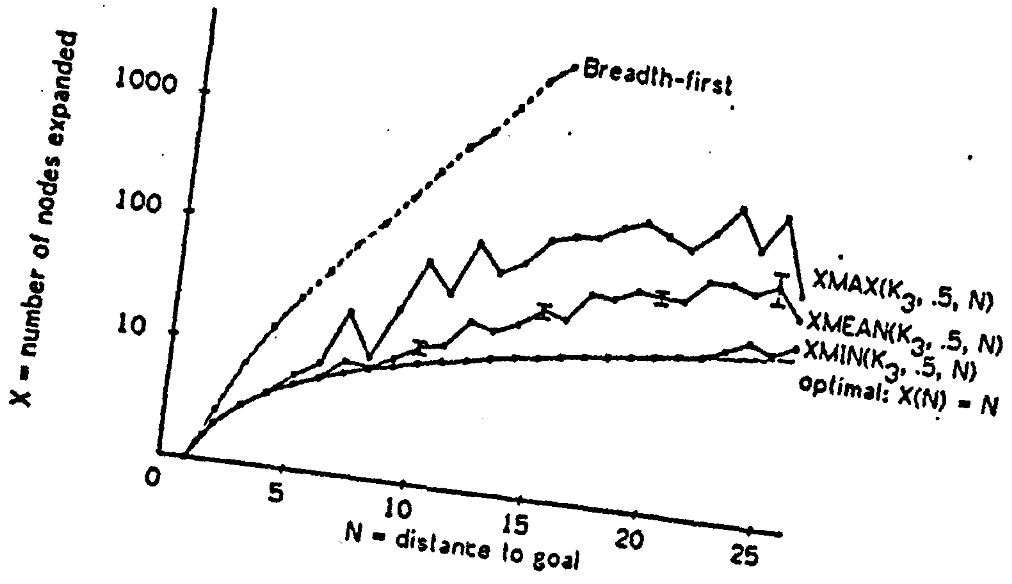


Figure 6.3a 8-Puzzle

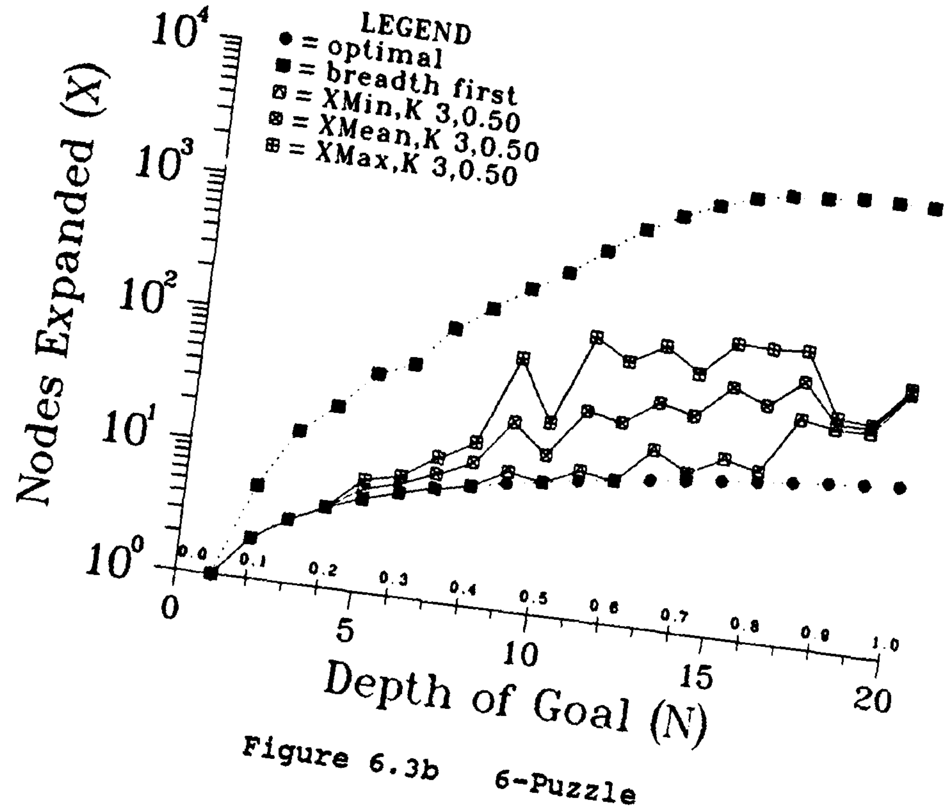


Figure 6.3b 6-Puzzle

Figure 6.4
XMEAN vs N
Heuristics K1, K2, K3
W = 0.5

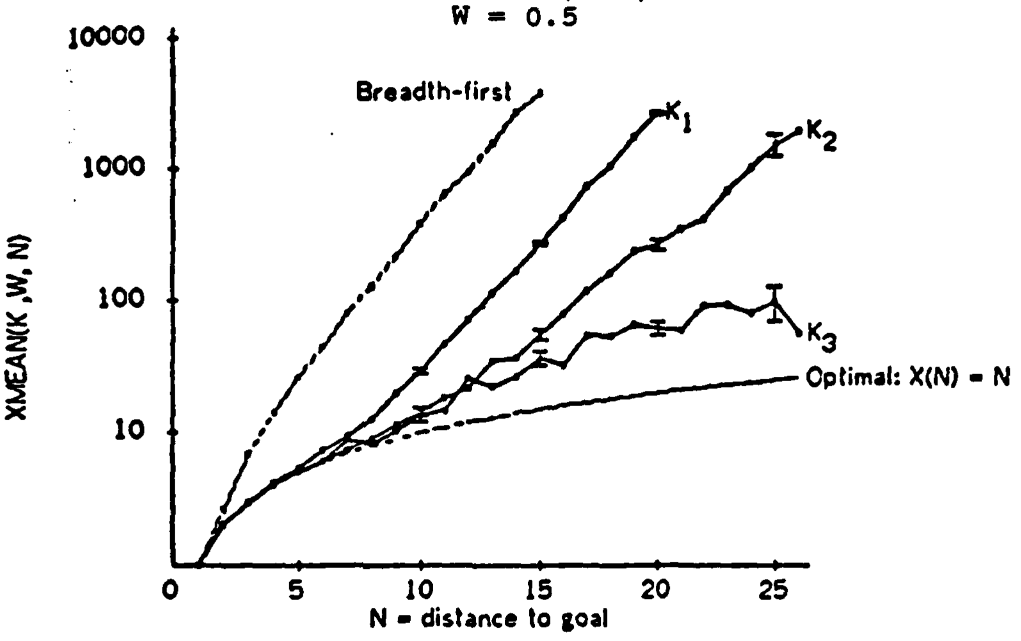


Figure 6.4a 8-Puzzle

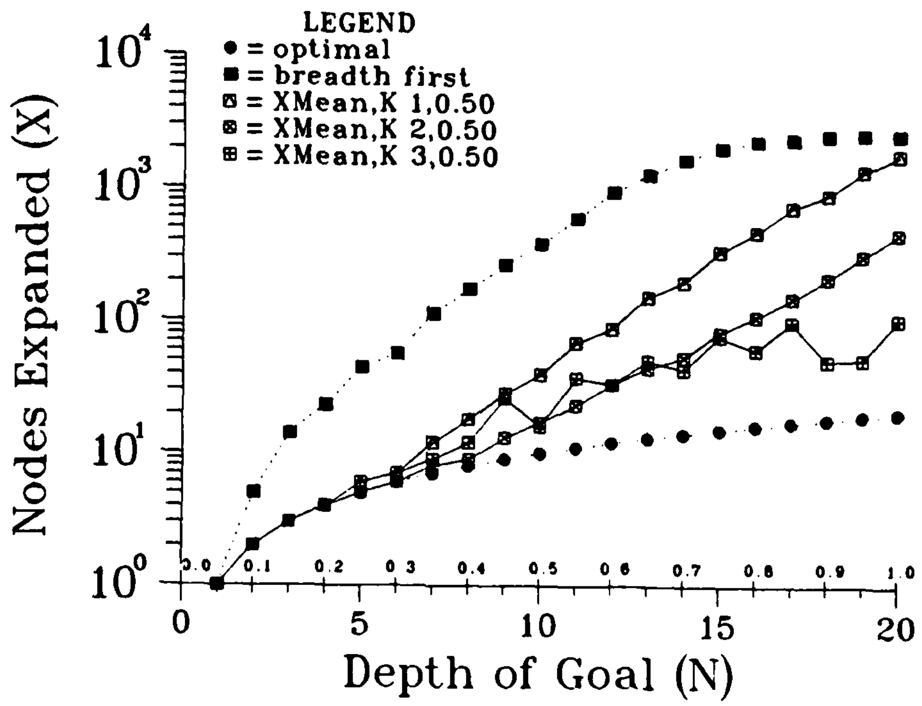


Figure 6.4b 6-Puzzle

Figure 6.5
LMIN, LMEAN, LMAX vs N
Heuristic K3
W = 0.5

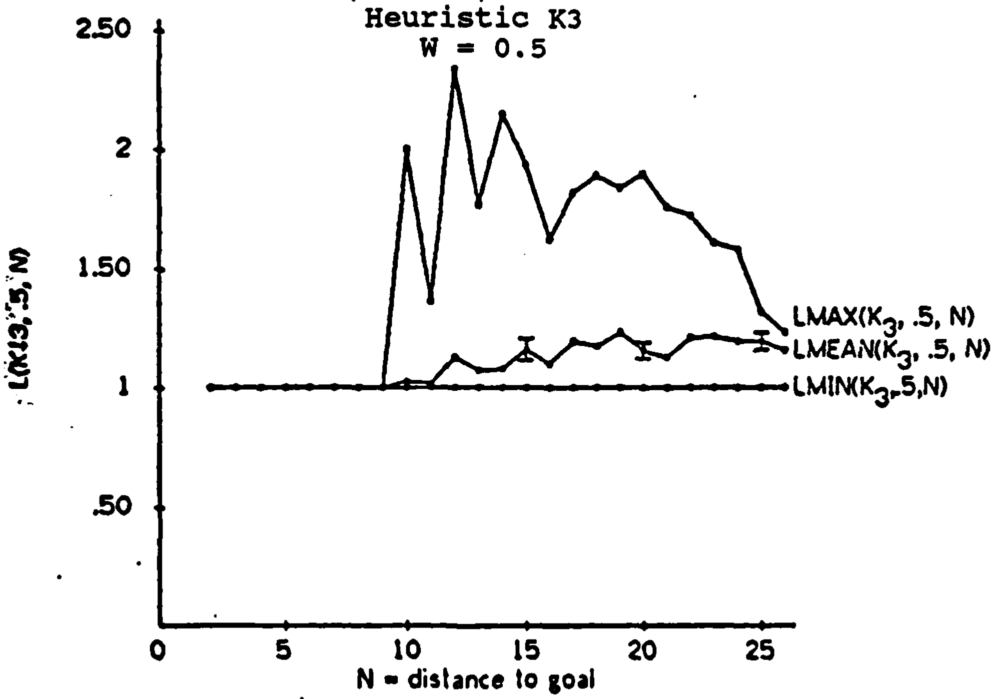


Figure 6.5a 8-Puzzle

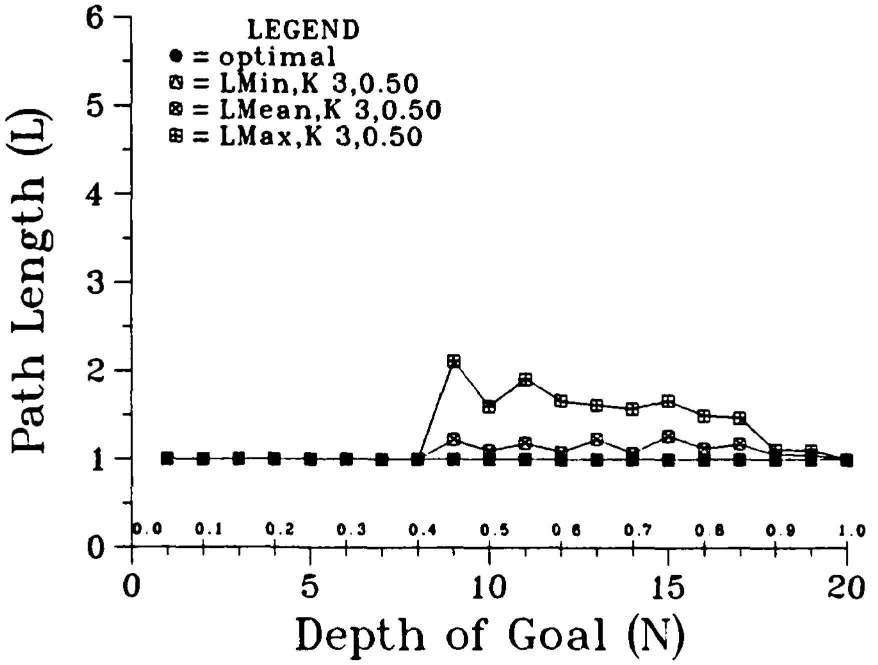


Figure 6.5b 6-Puzzle

Figure 6.6
XMEAN vs N
Heuristic K1
W = 0.2, 0.5, 0.7, 1.0

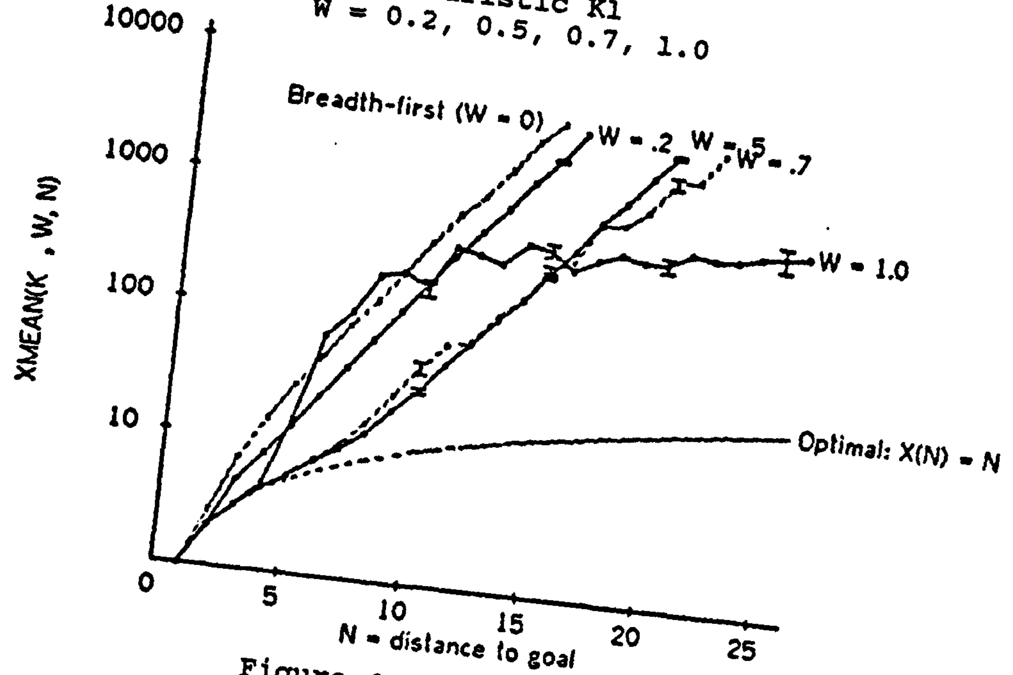


Figure 6.6a   8-Puzzle

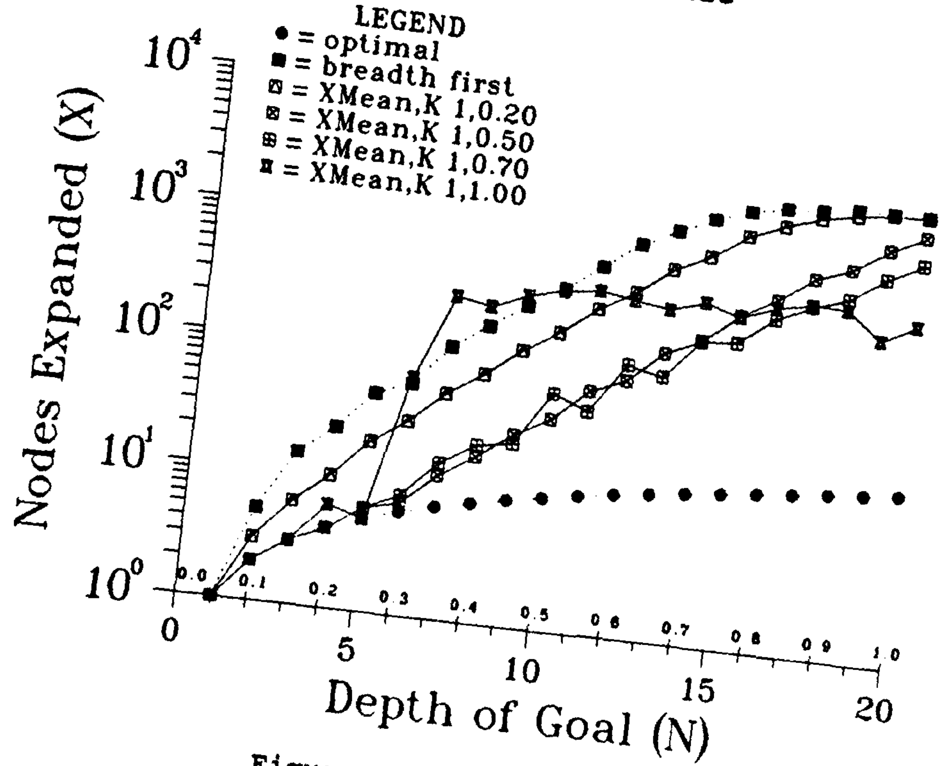


Figure 6.6b   6-Puzzle

Figure 6.7
XMEAN vs N
Heuristic K2
W = 0.2, 0.5, 0.7, 1.0

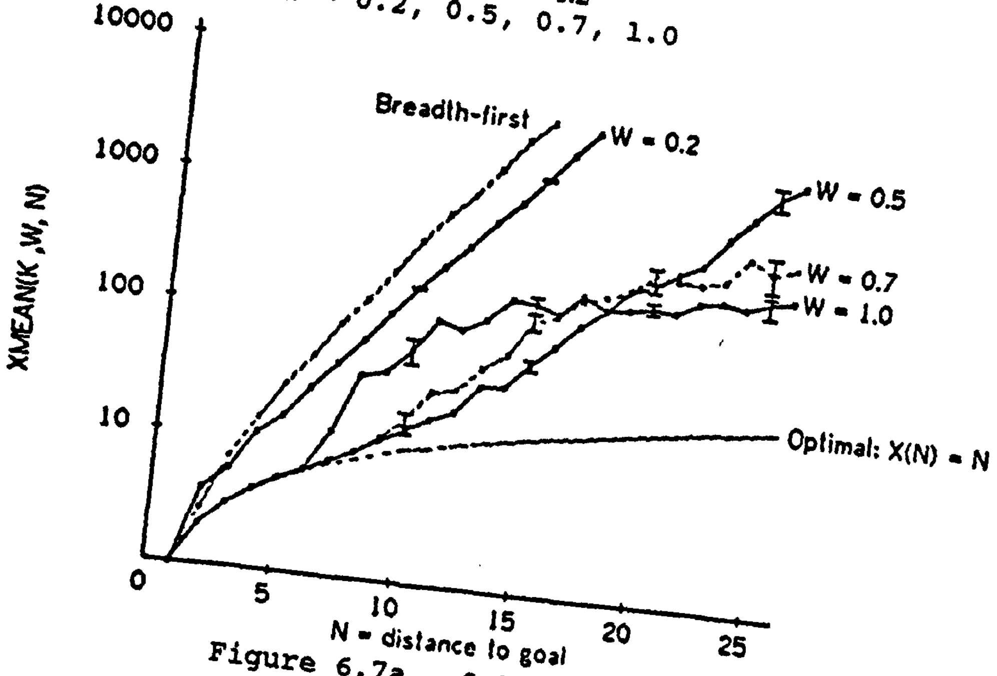


Figure 6.7a 8-Puzzle

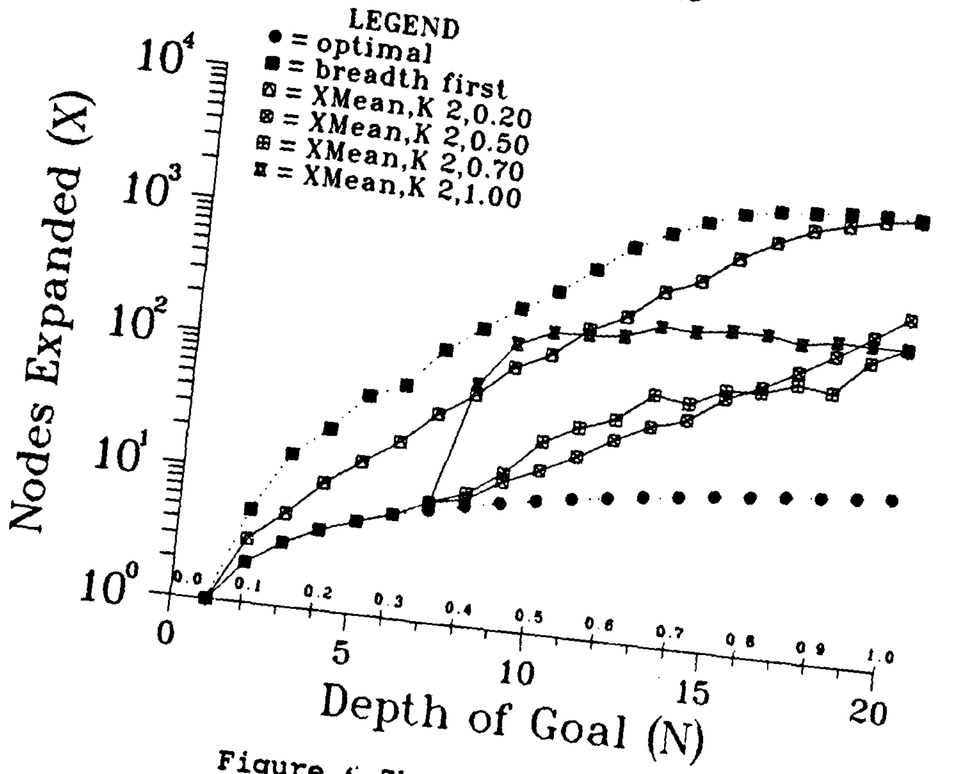


Figure 6.7b 6-Puzzle

Figure 6.8
XMEAN vs N
Heuristic K3
W = 0.2, 0.5, 0.7, 1.0

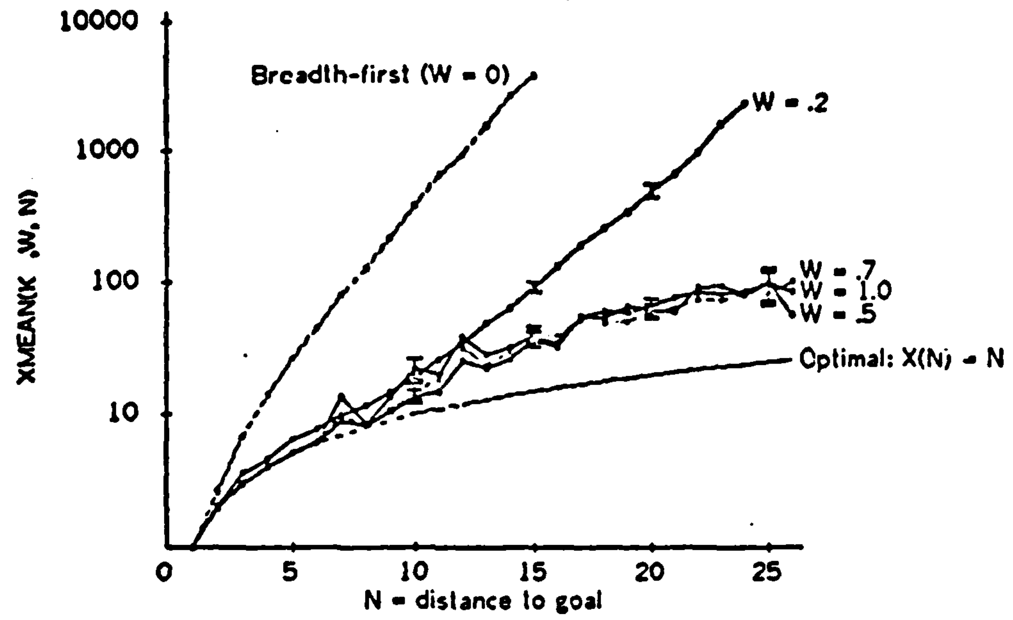


Figure 6.8a 8-Puzzle

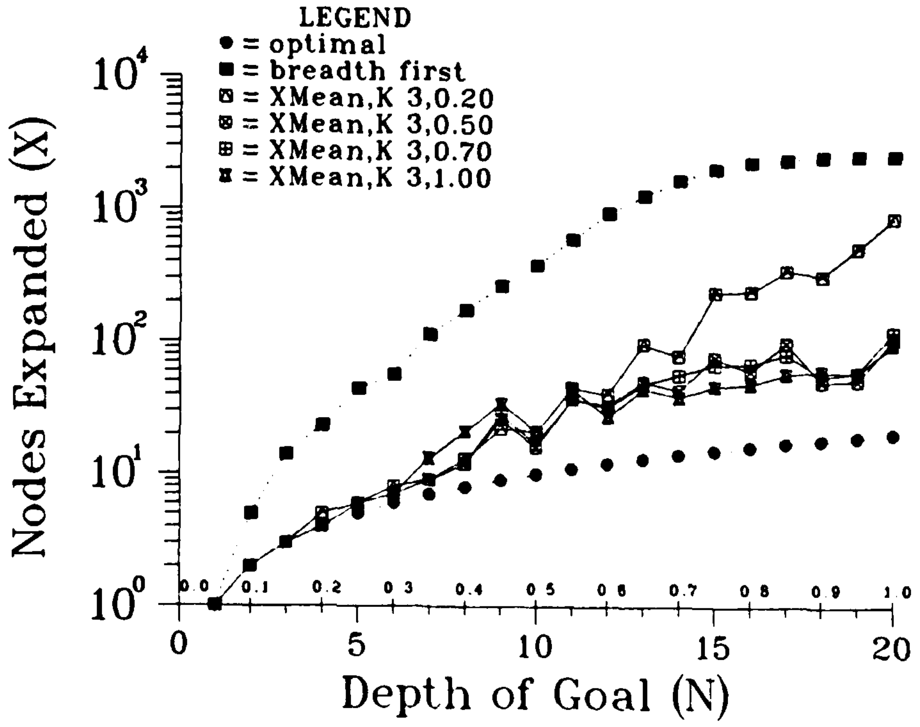


Figure 6.8b 6-Puzzle

Figure 6.9
XMEAN vs W
Heuristic K2
various N

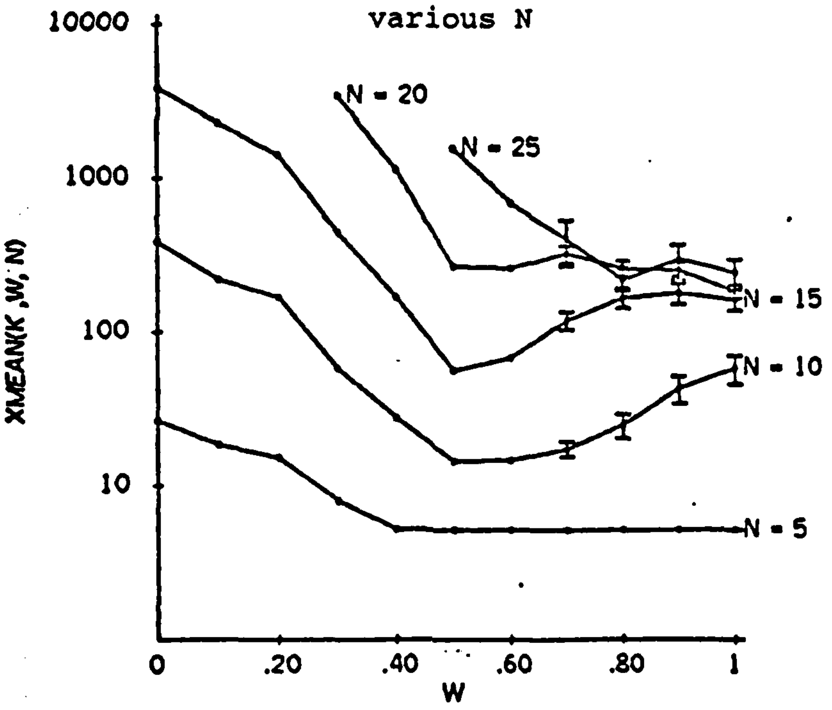


Figure 6.9a 8-Puzzle

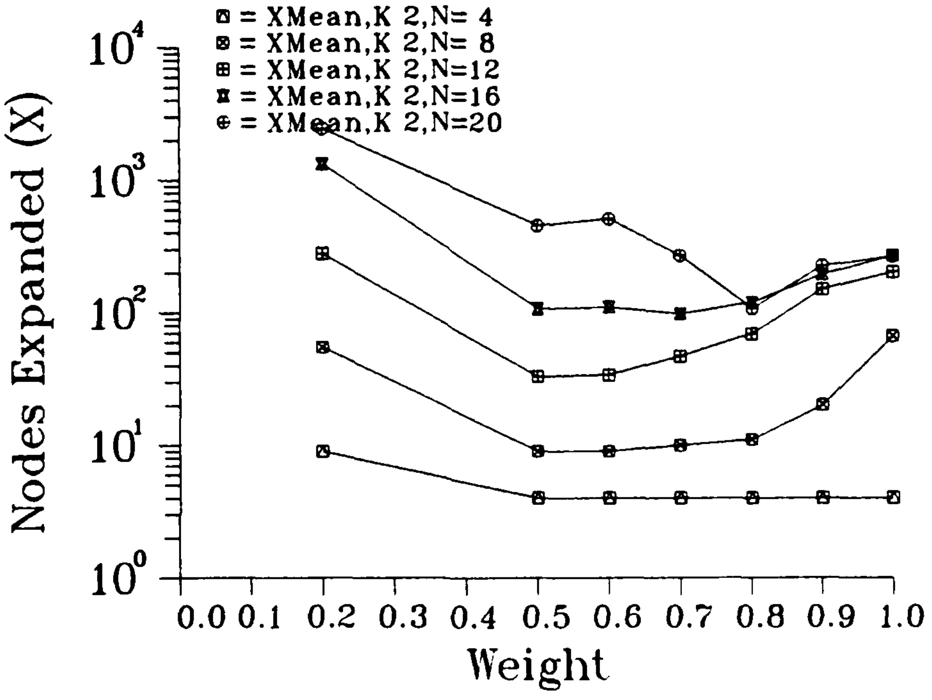


Figure 6.9b 6-Puzzle

Figure 6.10
XMEAN vs N
Heuristics K1, K2, K3
W = 1.0

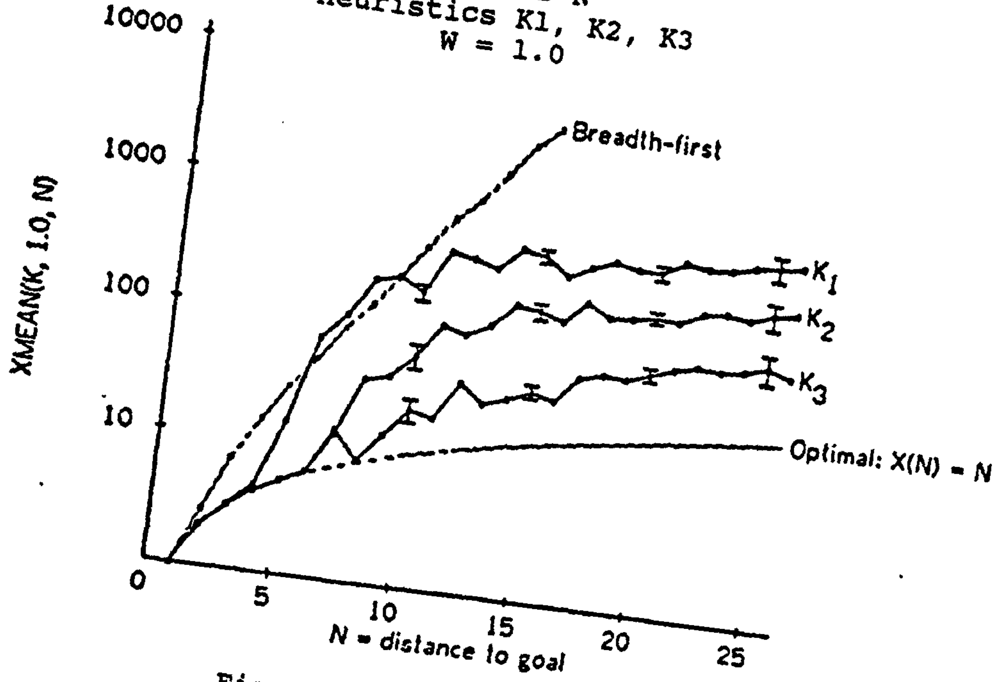


Figure 6.10a 8-Puzzle

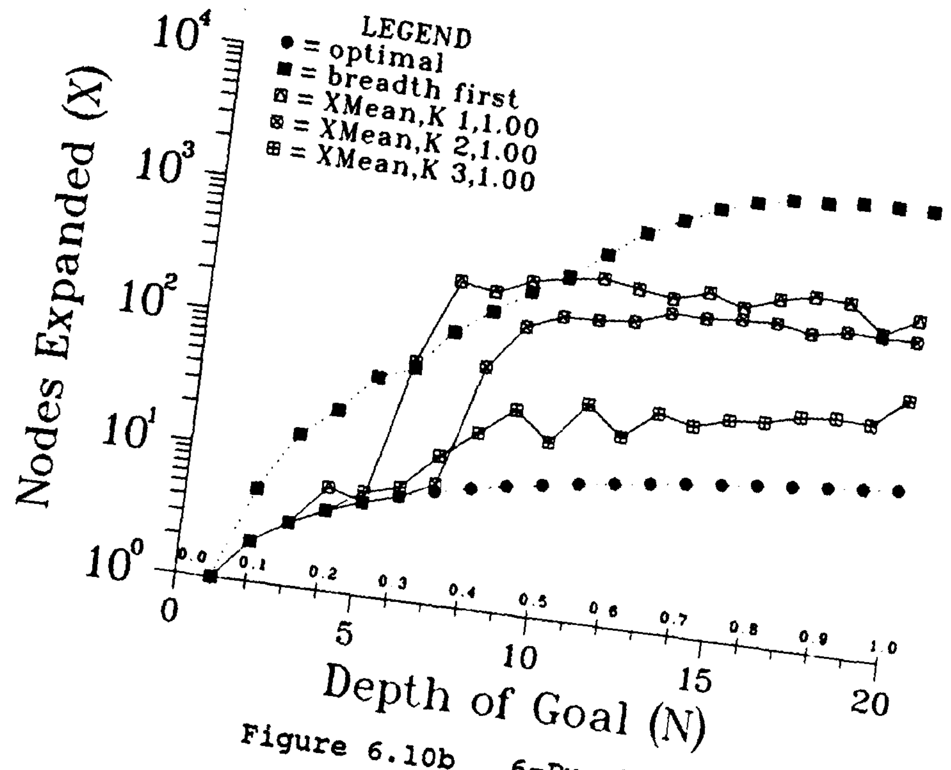


Figure 6.10b 6-Puzzle

Figure 6.11
LMEAN vs N
Heuristic K1
W = 0.7, 0.8, 0.9, 1.0

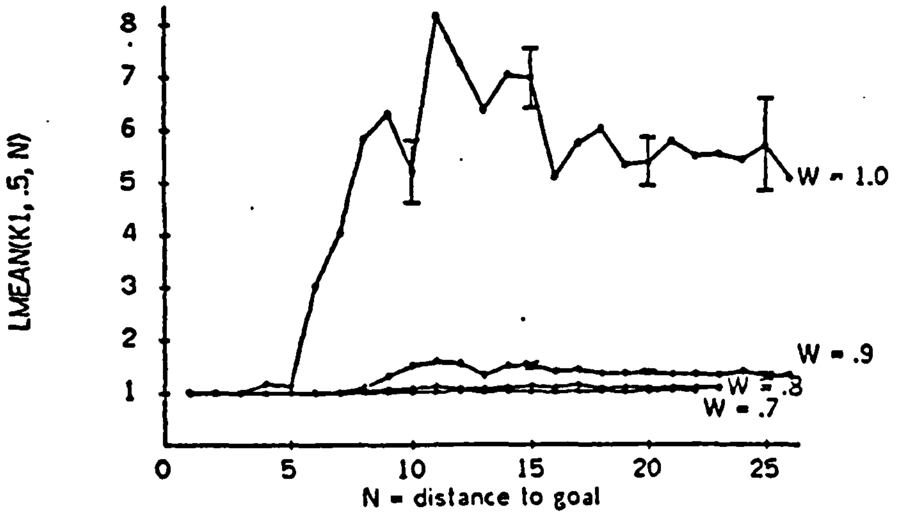


Figure 6.11a 8-Puzzle

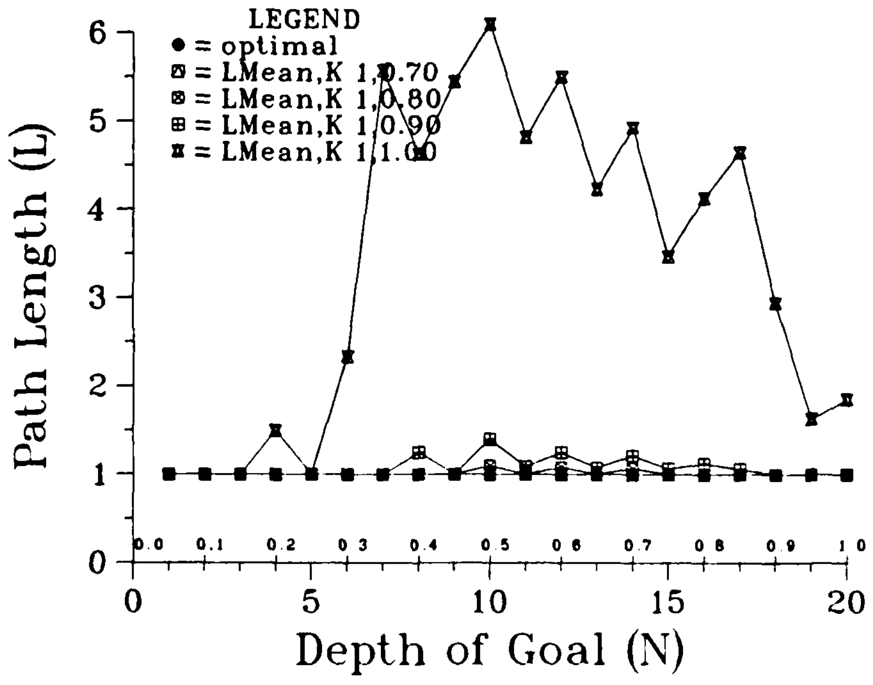


Figure 6.11b 6-Puzzle

Figure 6.12
LMEAN vs N
Heuristic K2
W = 0.7, 0.8, 0.9, 1.0

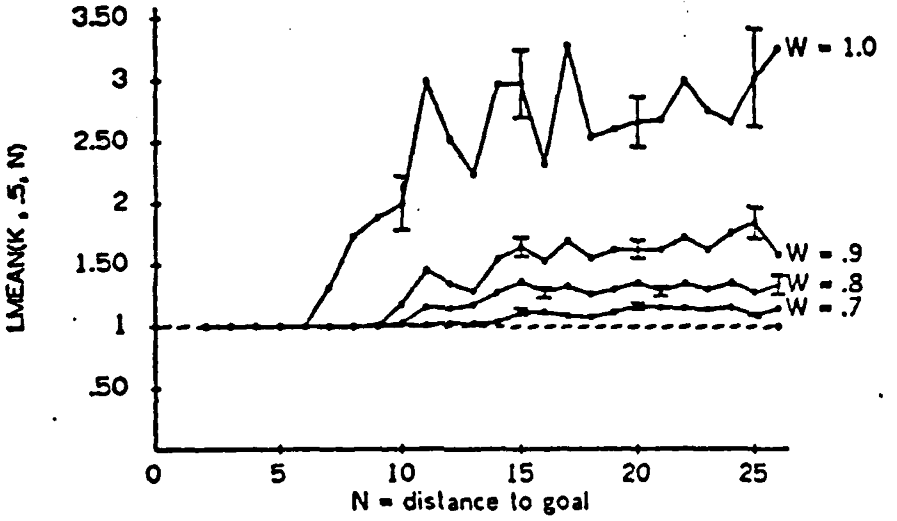


Figure 6.12a 8-Puzzle

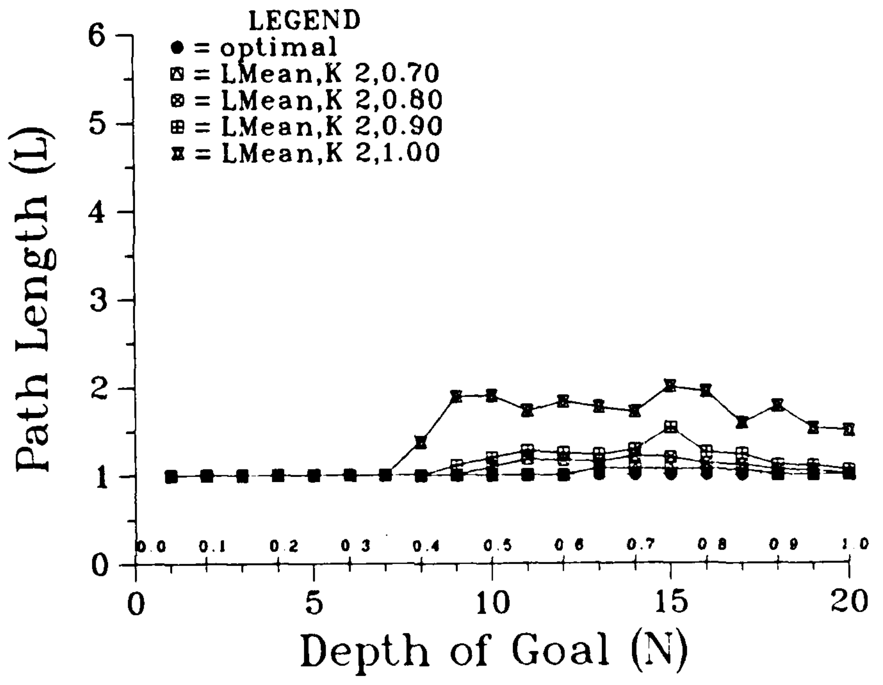


Figure 6.12b 6-Puzzle

Figure 6.13
LMEAN vs N
Heuristic K3
W = 0.5, 0.7, 0.9, 1.0

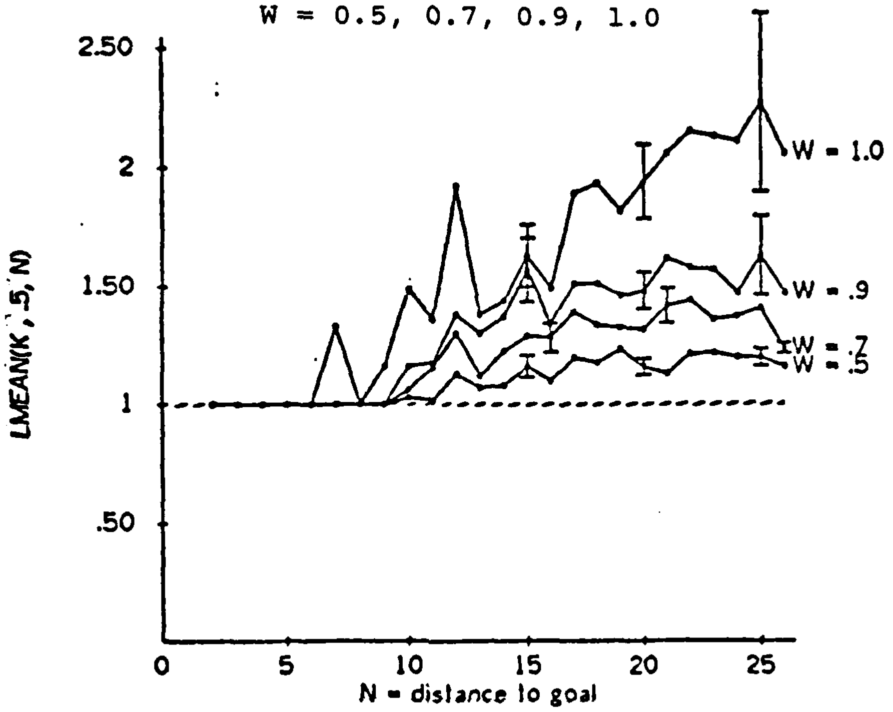


Figure 6.13a 8-Puzzle

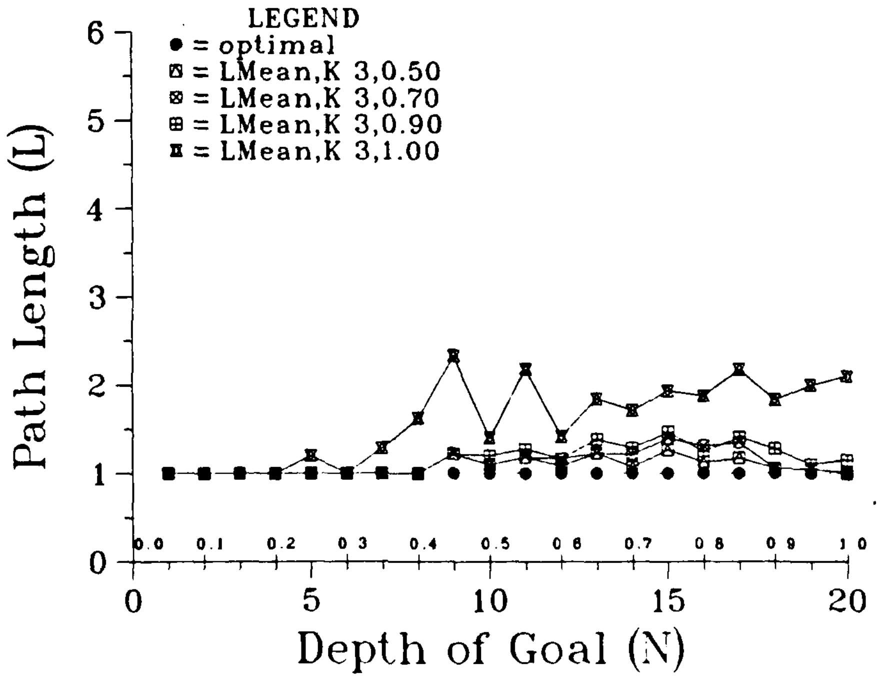


Figure 6.13b 6-Puzzle

Figure 6.14
LMEAN vs N
Heuristics K1, K2, K3
W = 1.0

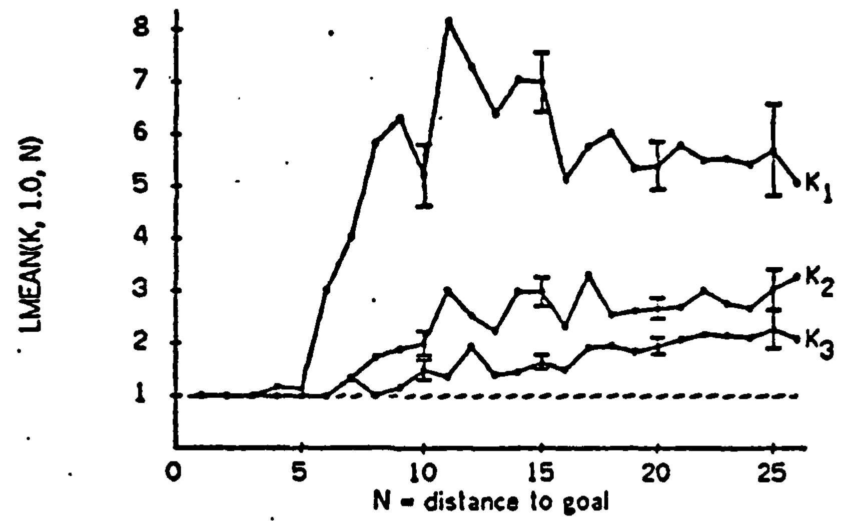


Figure 6.14a 8-Puzzle

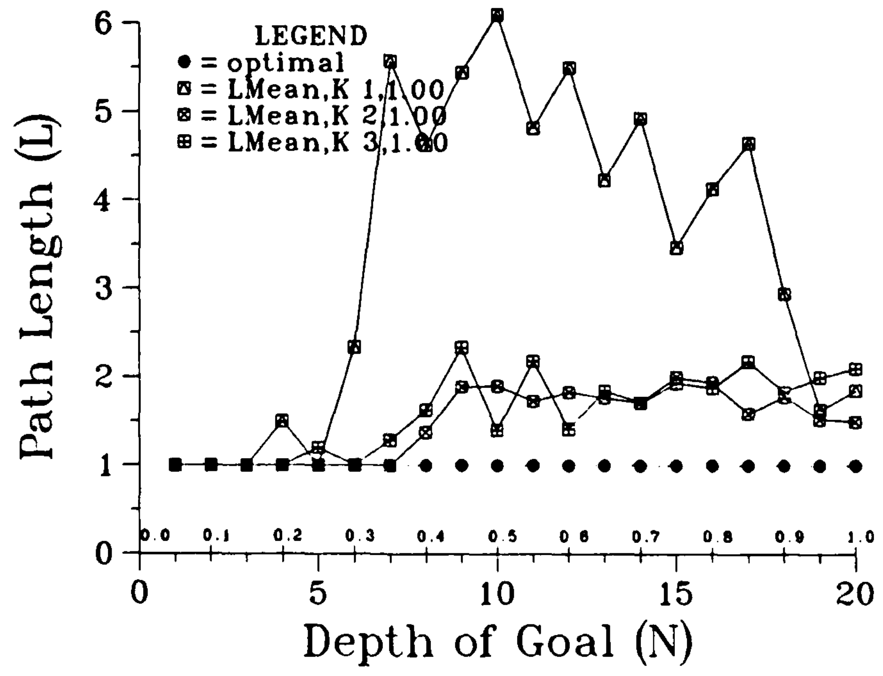


Figure 6.14b 6-Puzzle

Figure 6.15
LMEAN vs W
Heuristic K1
various N

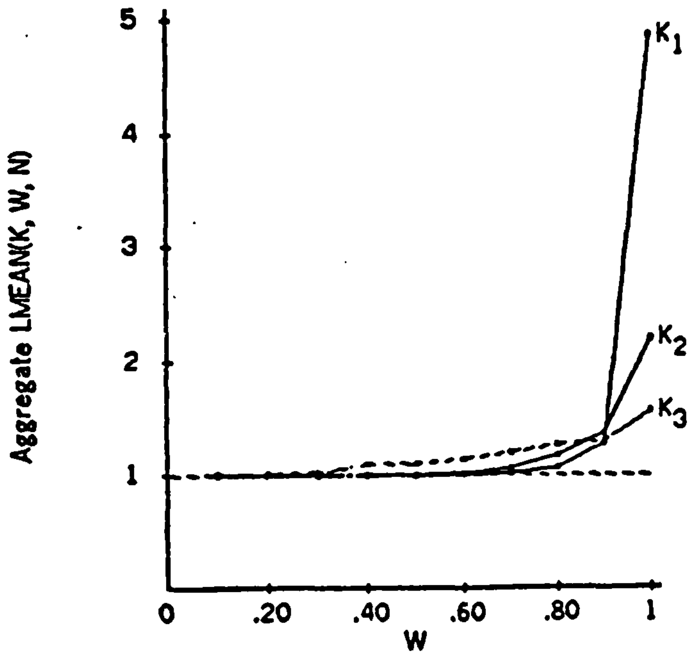


Figure 6.15a 8-Puzzle

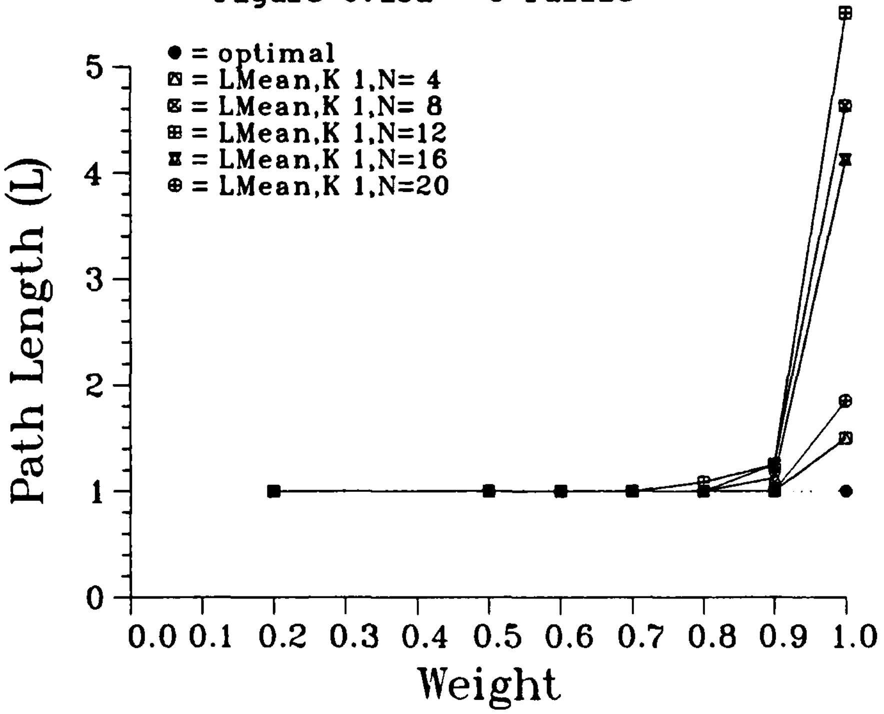


Figure 6.15b 6-Puzzle

Figure 6.16
LMEAN vs W
Heuristic K2
various N

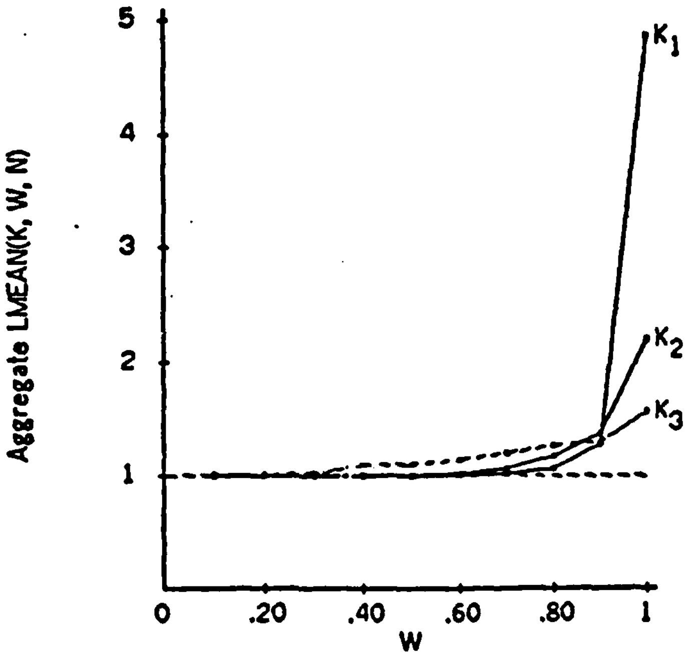


Figure 6.16a 8-Puzzle

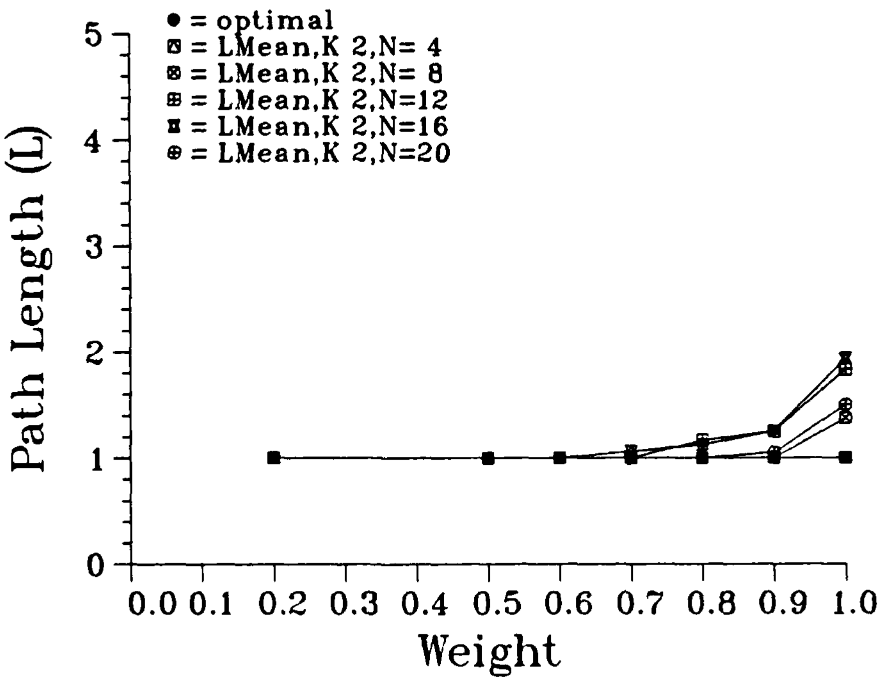


Figure 6.16b 6-Puzzle

Figure 6.17
LMEAN vs W
Heuristic K3
various N

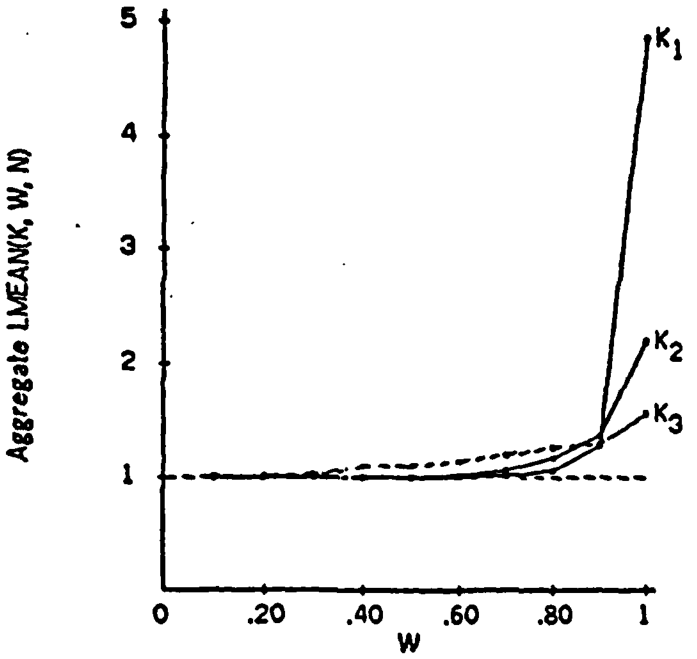


Figure 6.17a 8-Puzzle

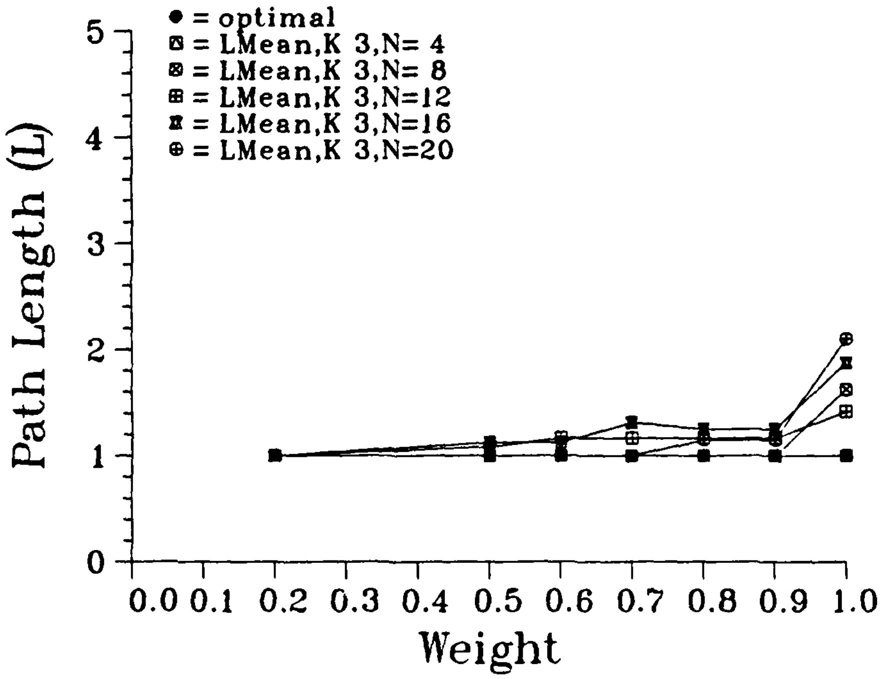


Figure 6.17b 6-Puzzle

Figure 6.18
XMAX vs N
Heuristic K2
W = 0.2, 0.5, 0.7, 1.0

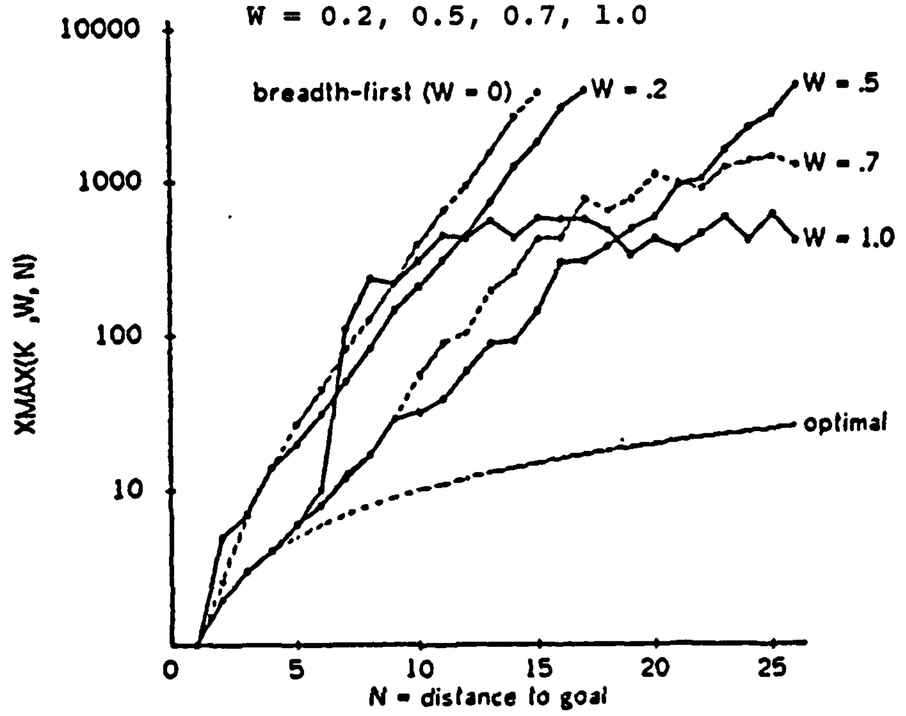


Figure 6.18a 8-Puzzle

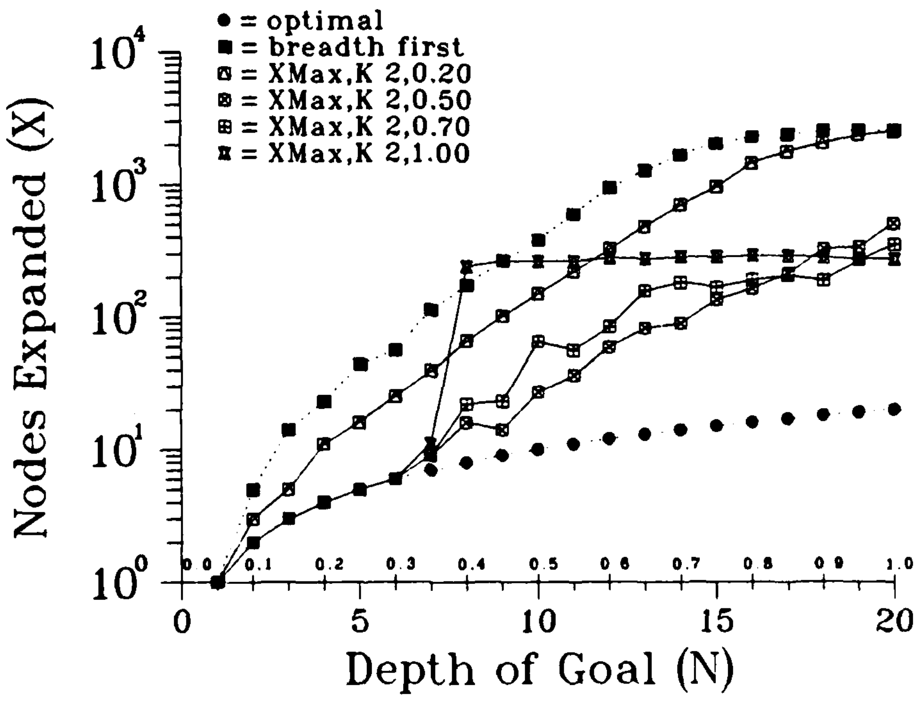


Figure 6.18b 6-Puzzle

Figure 6.19
KMIN, KMEAN, KMAX vs I
Heuristic K1

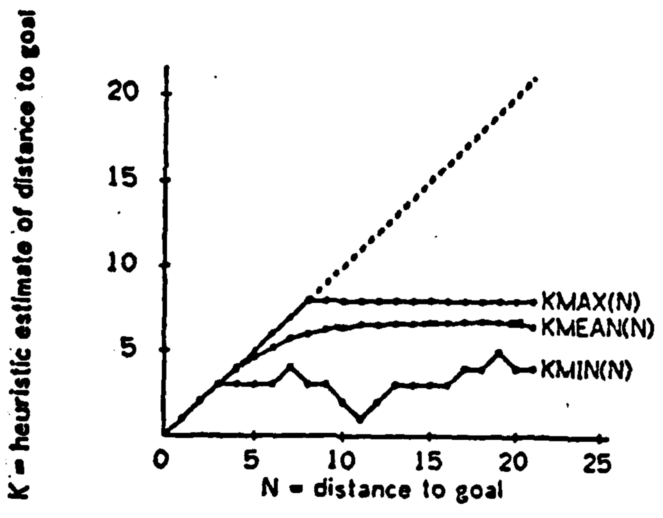


Figure 6.19a 8-Puzzle

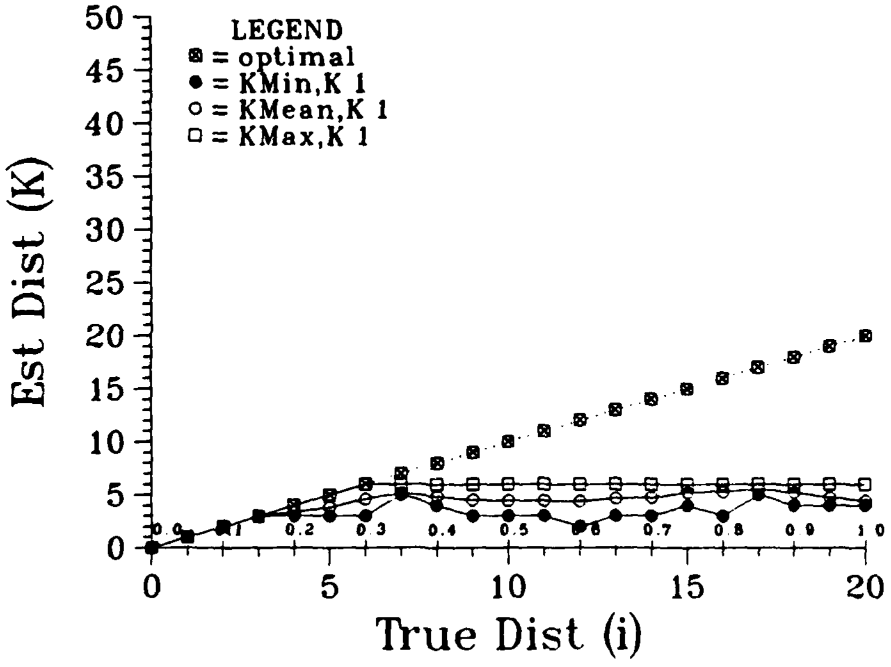


Figure 6.19b 6-Puzzle

Figure 6.20
KMIN, KMEAN, KMAX vs I
Heuristic K2

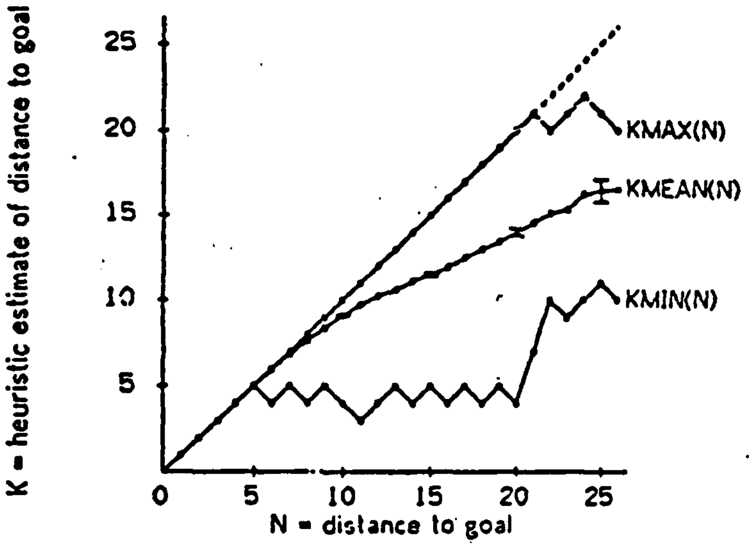


Figure 6.20a 8-Puzzle

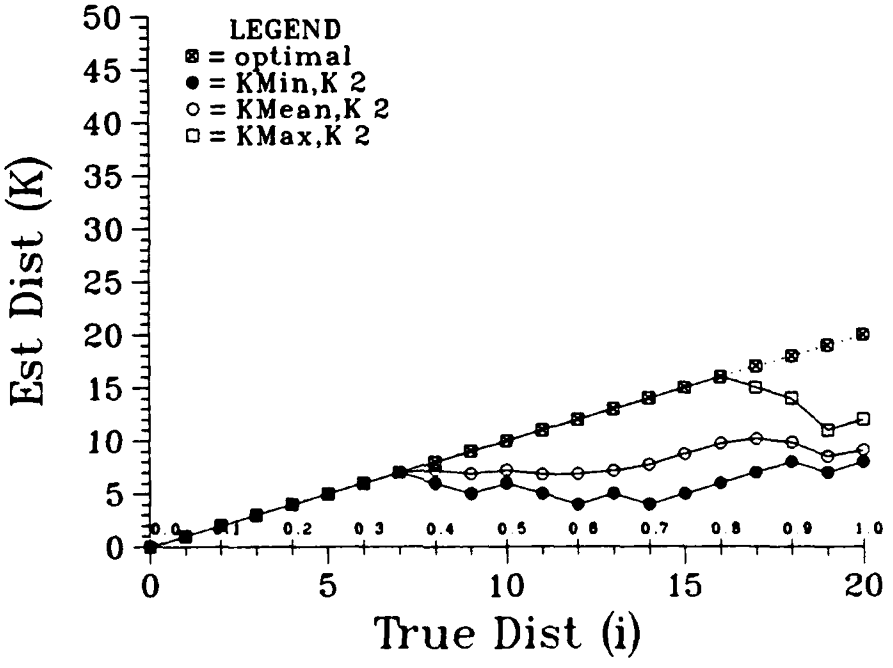


Figure 6.20b 6-Puzzle

Figure 6.21
KMIN, KMEAN, KMAX vs I
Heuristic K3

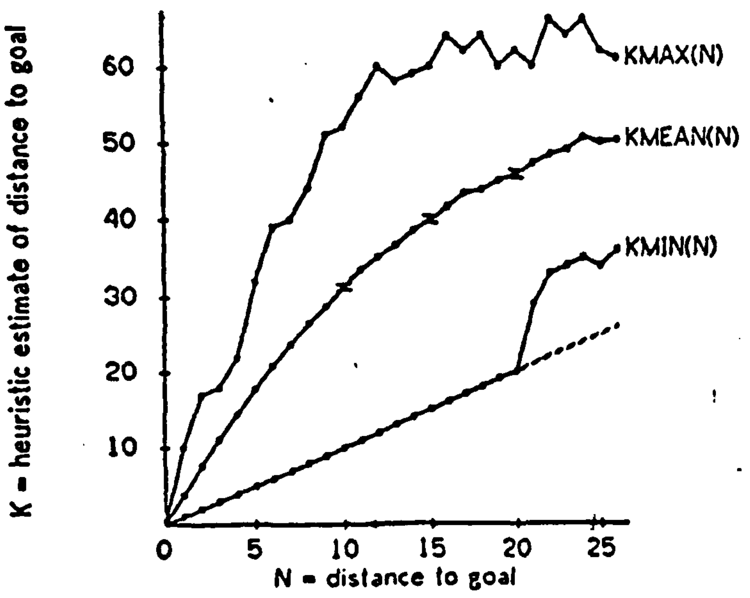


Figure 6.21a 8-Puzzle

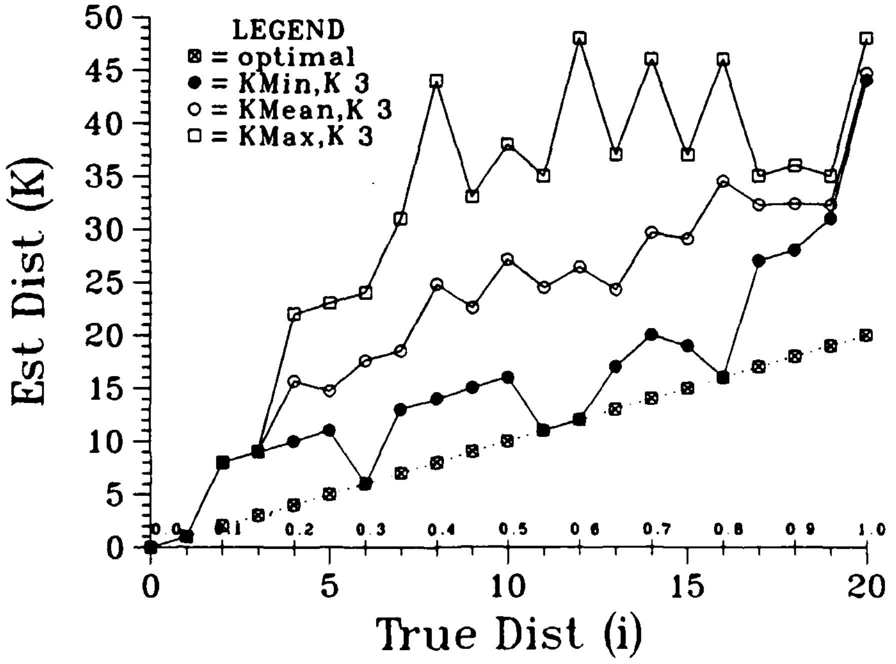


Figure 6.21b 6-Puzzle

1. DISCUSSION

We found amazing agreement between the performances of our K1 and K2 heuristics and Gaschnig's K1 and K2 for nodes expanded at every weight. Our K3 was similar to Gaschnig's K3, but is not as close in agreement as evidenced when compared to K2 in Figures 6.4 and 6.10. In both of these figures, Gaschnig's K2 produced consistently poorer results than K3, but in our implementation, this is not always the case.

Another minor inconsistency was that the solution path lengths that our heuristics discovered were slightly better than Gaschnig's, although the relative shapes of the curves are roughly the same. We feel this is because of the difference in size of the state spaces of the two domains, making it inherently easier to find longer solution paths in the 8-Puzzle than in the 6-Puzzle. However, Figure 6.14 shows that the path length for our K2 and K3 are inter-twined, while Gaschnig's K2 and K3 were distinctly tiered.

D. CONCLUSION

We hope the reader was as impressed with the similarity between the two domains as we were. The results were similar enough to conclude that changing the domain had little effect on these heuristics, and to say that the 6-Puzzle is a close cousin to the 8-Puzzle. It would be interesting to apply the experiments conducted in this chapter to other Beads World variations to see if the observed similarity is shared on a broader scale than just between the 6-Puzzle and the 8-Puzzle.

VII. BEADS WORLD PROGRAM TOOLS

A. INTRODUCTION

Discovery of the Beads World as a very rich and convenient puzzle domain in which to investigate various aspects of heuristic search added an additional set of objectives to the original objectives of this combined research effort. In addition to the original research described elsewhere in both of these theses, it was felt that one of the most important contributions that this project could make would be the development and implementation of a set of generalized, reusable, convenient programming tools supporting research in the Beads World domain. In addition to providing a very useful environment in which to carry on our own investigations, these tools will hopefully allow other researchers to extend our work and to investigate the many areas left untouched by our work without having to go through the lengthy process of developing their own program tools. This section of these documents serves as both the complete documentation of the program tools developed and as a description of their application and use.

These program tools were developed in VAX Pascal, and in one case VAX Fortran. They are currently running on the

RICC VAX-11/780, but again with one exception, are not necessarily dependent on the VAX/VMS environment. These tools are generalized -- in other words, with the proper configuration of a few global domain-description variables, these tools automatically reconfigure themselves to work with any of the Beads World configurations. No recoding is necessary -- all reconfiguration may be accomplished by changing input data to the application programs.

A fair amount of attention was given to applying sound software engineering principles in the implementation of these tools. As currently implemented the software is comprised of well over 3000 lines of Pascal and FORTRAN code. In order to reduce program source modules to a manageable size, and to avoid the duplication of code across many similar but different applications, the module facility provided by VAX Pascal has been used extensively. These modules are somewhat separable -- the user only need link with those modules which contain data structures or procedures which must be imported for the particular application. (As it turns out, these modules, although cleanly, functionally separated, are so tightly interrelated in their overall operation that they are almost always all necessary.) Inside these modules, data structures and procedures have been packaged in their cleanest possible form. This, combined with the logical organization imposed by the module structures, makes the program code almost self-documenting, allowing convenient maintenance and enhancement of this package by future users.

From the AI researcher's perspective, the tools in this package can be divided into three different functional categories. Some of the tools are involved with investigating the graph characteristics of various Beads World configurations: expansion of a complete graph, analysis of its characteristics, enumeration of its elements, and the generation of a representative sample of its nodes. Most of the tools are involved with investigating A* heuristic search, providing flexibility as to which control structures and heuristics are used, as well as in the type of data that is gathered. A third and somewhat separate set of tools is involved with displaying the data generated by the routines of the second category in a convenient and meaningful graphic form.

From the program organization perspective, this package can again be divided into three functional categories. The first consists of several modules which provide general utility functions, control structures, heuristics, and statistical aids which can be used by a variety of applications. The second category consists of a set of applications modules for investigating search spaces, profiling heuristics, and solving Beads World puzzles to collect performance data on heuristic search techniques. The third module is again separate from the other two, and is responsible for the graphic output of data.

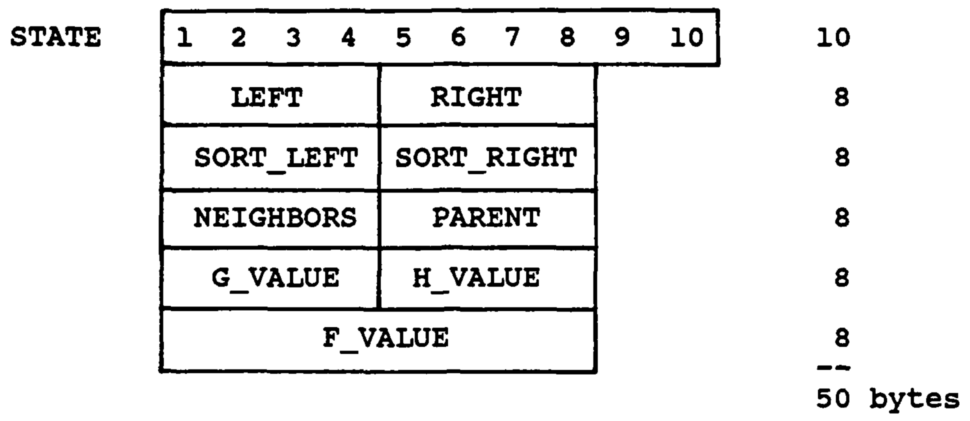
The following sections describe the various utility modules and applications programs, both in terms of their structure and operation (where appropriate for understanding the tools and their behavior), and in terms of their use by future researchers who wish to use the applications provided or to create their own applications. The source code is listed in Appendices A through C and is also included on the resource diskette provided with this document, as detailed in Appendix D.

B. DATA STRUCTURES -- PUZZLE AND GRAPH REPRESENTATION

Before describing the structure and function of the various utility procedures and programs provided and the use of these in other applications programs, it will first be useful to describe the fundamental data structures used throughout the package. These represent unique Beads World puzzle states, the graph which represents all of the possible states in a Beads World configuraton and their relationships, and the search tree which is created during the performance of a search between two states of this graph.

Figure 7.1 illustrates the structure of the most fundamental unit, the puzzle node descriptor. The first field describes the state of the puzzle, or the particular locations of each of the tiles or beads and the blank or

Figure 7.1 Puzzle node record structure.



blanks. Tile positions are always numbered, with 1 identifying the center position and 2 through n identifying, in order, clockwise positions about the center of the puzzle. Tiles are also numbered, with 0 indicating a blank and 1 through n-1 uniquely indicating each of the beads. STATE is a packed character array with space for puzzles with up to 10 positions. Each element of the array corresponds to a location, and the character content of each element specifies the tile located at that position. The use of a packed array for describing puzzle states saves space and makes state comparisons convenient.

The next four fields of the puzzle node record are the links used for list and tree maintenance by the various graph production and puzzle solution routines. LEFT and RIGHT are used to maintain doubly-linked lists of puzzle nodes; SORT_LEFT and SORT_RIGHT are used to maintain binary trees of puzzle nodes.

The next field, NEIGHBORS, is a pointer to a list of neighbor pointers -- this is the structure which links puzzle nodes into a graph. Neighbor nodes (illustrated in Figure 7.2(a)) are simply elements of a singly-linked list of pointers to other puzzle nodes. A node's neighbor list is then a list of all nodes which may be obtained by performing a single state transformation operation on the state of that node. Neighbor lists are used during creation of the graph associated with a Beads World configuration, and also by the graph search version of A* used for solving Beads World problems.

The PARENT pointer is used to construct a search tree as the A* algorithm explores a subset of a Beads World graph. Figure 7.2(b) illustrates the search tree structure. Note that although once fully constructed, neighbor lists do not change, a node's parent pointer may change several times as alternate solution paths are explored. Figure 7.2(c) illustrates the complete graph for the "3-puzzle" and the start and goal states for the search tree of Figure 7.2(b).

The final three fields of the puzzle node description record are used primarily during the A* search procedure. G_VALUE contains that node's current distance from the starting node. H_VALUE contains the current heuristic's estimate of that node's distance from some goal node in the graph. F_VALUE contains the weighted composite of G_VALUE and H_VALUE.

Figure 7.2(a) Neighbor node list.

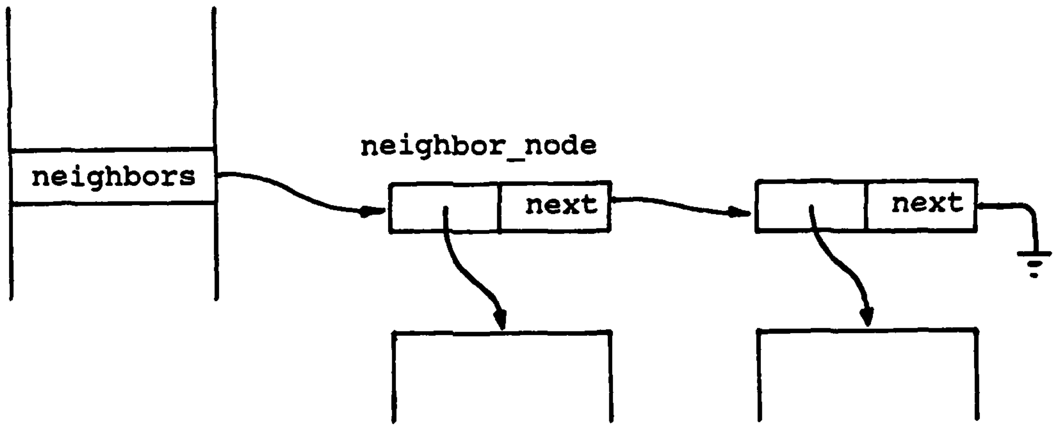


Figure 7.2(b) Example search tree for the 3-puzzle.

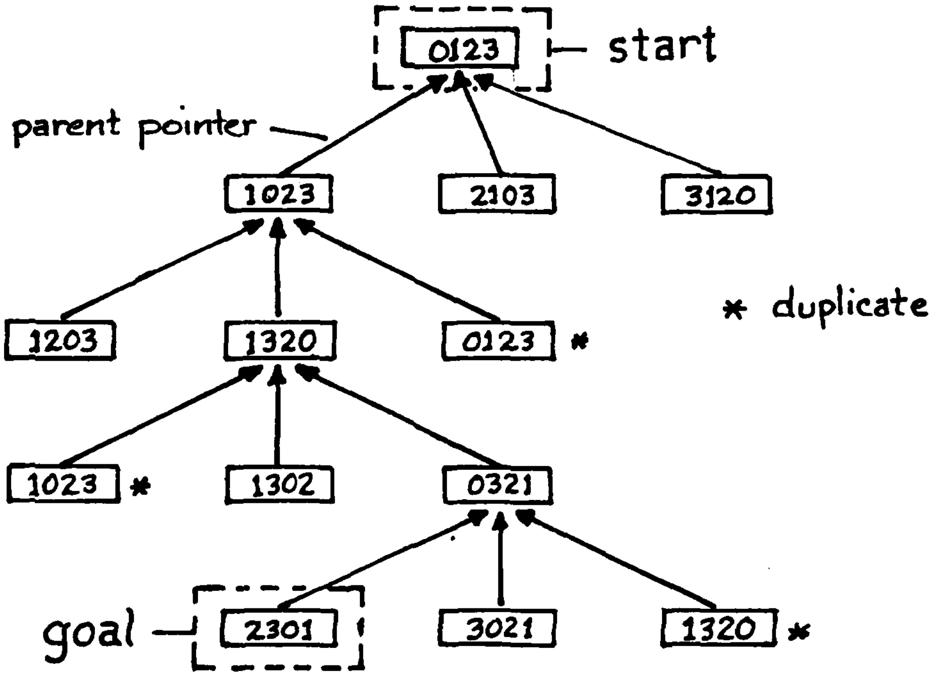
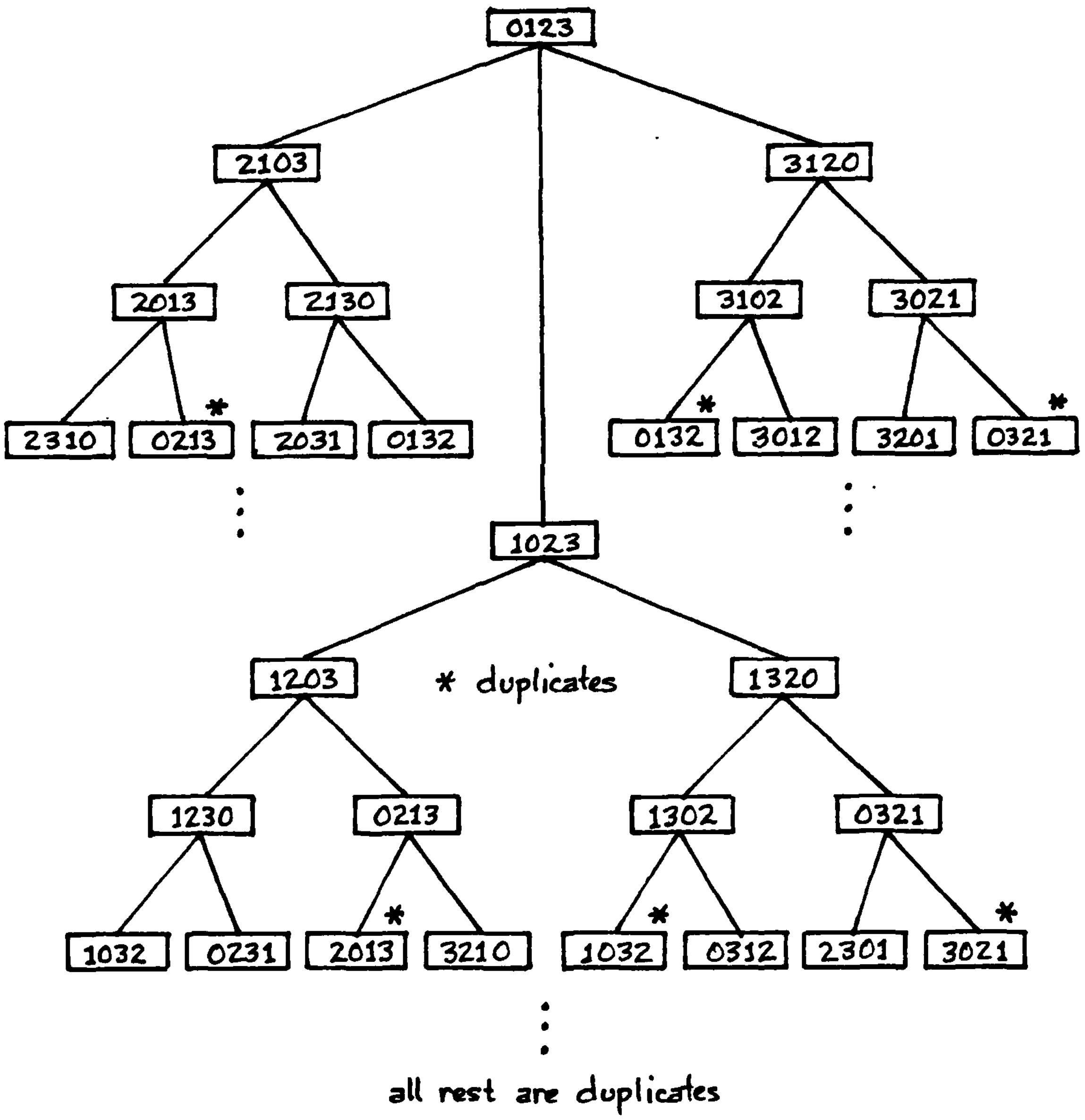


Figure 7.2(c) Example graph for the 3-puzzle, rooted at state (0 1 2 3).



C. GENERALIZED TOOLS MODULES

1. UTILITIES

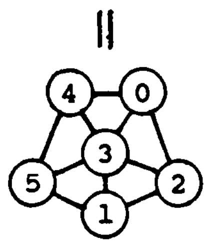
The utilities module packages a variety of functions and procedures for manipulating the data structures described in the previous section. Among these are routines handling dynamic allocation and deallocation, linked list manipulation, tree manipulation, and the input and output of puzzle state descriptions. The source code for this module is listed in Appendix A. Each of the routines is described below.

There are two procedures which provide a convenient way to read and write puzzle state descriptions. Procedure READ_STATE reads from the standard input a puzzle description for a puzzle of size n into a packed puzzle state array. The format for this puzzle description is shown in Figure 7.3. PRINT_STATE accepts an n-puzzle state description and writes it to the standard output in the same format.

Figure 7.3 Puzzle description for a state of the 5-puzzle.

bead numbers (3 4 0 2 1 5)

[positions 1 2 3 4 5 6]



CREATE_PUZZLE_NODE dynamically allocates a new puzzle descriptor node with the desired state and initializes all of the other fields to NIL or 0 as appropriate. The complementary procedure FREE_NODE deallocates puzzle nodes, but in addition, systematically frees the neighbor node elements of that node's neighbor list.

Linked lists of puzzle nodes are used throughout the various modules which make up this package. All of these lists are doubly-linked, and have a header node whose G_VALUE is a count of the number of elements on the list. Lists in the program code are then simply pointers to the header nodes of these doubly-linked lists. Several functions and procedures are provided for manipulating these lists. CREATE_EMPTY_LIST returns a pointer to a header node with no list elements. IS_EMPTY is a boolean function which returns a TRUE value if the LIST has no nodes other than the header. PLACE_ON_END_OF_LIST adds a node to a list in a queue-like manner.

PLACE_IN_ASCENDING_ORDER is a special procedure (used by A* in ordering OPEN lists) which places a node on a list by ascending F value. This routine resolves ties in the F value by placing the most recent node with some F value before all of the other nodes with that same value. As was discussed in a previous section, this can have an impact on the performance of A*; the tie-resolution convention can be easily changed by a small change inside this routine. REMOVE_FROM_FRONT_OF_LIST performs the expected operation, returning a pointer to the first node on the list and removing it from the list. DELETE_FROM_LIST removes the specified node from anywhere in the list. Finally, FREE_LIST deallocates all of the nodes on LIST and then disposes of the header.

Our implementations of A* use binary trees to keep track of nodes already generated. Three routines are provided which allow the use of the binary tree structure. INSERT_IN_TREE inserts the node being pointed to in a tree in "inorder" fashion, keyed by the alphabetic order of the state representations held in the node and the nodes already on TREE. INSERT_IN_TREE does not attempt to balance the tree.

FIND_IN_TREE and FIND_STATE_IN_TREE perform essentially the same function. Given a pointer to a puzzle node, FIND_IN_TREE tries to find and return a pointer to the node in the tree having the same puzzle state. If no such node is found, FIND_IN_TREE returns NIL. FIND_STATE_IN_TREE returns a pointer to the node in TREE having the desired puzzle state.

Finally, two procedures are provided to allow the deallocation of the search tree and graph structures that are created out of puzzle nodes and their neighbor lists. FREE_BINARY_TREE recursively deallocates the nodes in a binary tree. FREE_GRAPH recursively disposes of all of the nodes and neighbor lists which make up a Beads World graph.

2. CONTROL STRUCTURES

This module contains the routines which perform graph space generation and which implement the two versions of the A* search alogorithm discussed in Section 5. In addition this module exports data structures which describe the characteristics of a graph space and the results of a search (and the routines which initialize these). Most of the routines provided by the utilities module are imported by the control module, as well as the highest level routine from the heuristics module described below.

Two routines are primarily involved with the generation of Beads World graphs. GENERATE_GRAPH accepts a starting puzzle state and generates a complete graph from this, returning three items. The first is a pointer to the inorder binary tree containing all of the puzzle nodes, ordered alphabetically by state descriptor. The second is a pointer to the generated graph structure. (Note that these pointers always point to the same node -- the one which contains the starting state.) The third is a graph descriptor record, whose structure is described below.

Although there is no start or end to the graph space associated with a particular Beads World configuration, as a practical consideration, the generation of this graph space has to start somewhere. The algorithm which GENERATE_GRAPH uses is similar to the basic shell of the A* algorithm, and is given in Figure 7.4. Unexplored

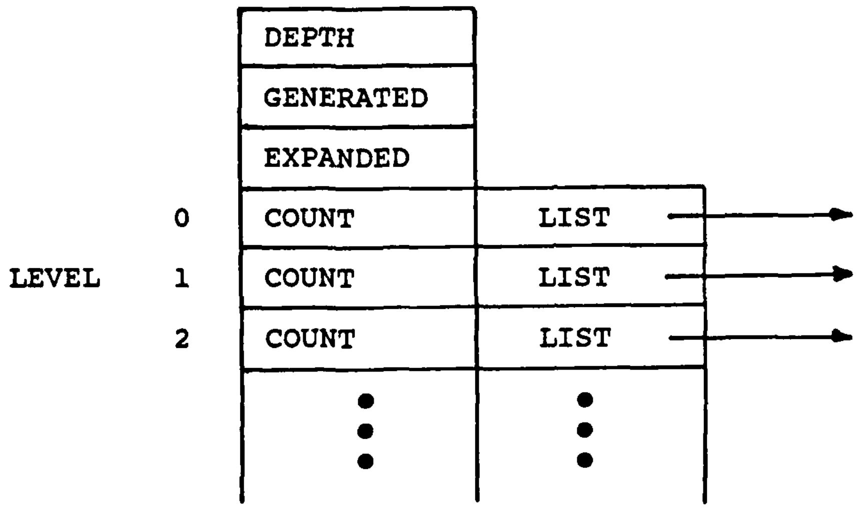
Figure 7.4 Graph generation algorithm.

```
place the starting node on the OPEN list
while unexplored nodes remain do
    remove one
    generate its successors
    for each successor do
        if it is a new state then
            add it to:
              the search tree
              the parent's neighbor list,
              and OPEN
        else
            add the original node to that
              parent's neighbor list
        end_if
    end_for
end_while
```

(unexpanded) graph nodes are maintained on an OPEN list, which is ordered by increasing "depth" in the graph (distance from the starting node). Nodes to be expanded are removed from the front of the list, which means that the graph generation proceeds in a breadth-first manner. When each node is removed for expansion, all of its successors are generated. Each successor is in turn examined. It is first assigned a G value (depth) one greater than its parent's; then the search tree is examined to see if this is a new node or a previously discovered one. If it is new, then it is added to the parent's neighbor list, to the search tree, and to OPEN. If it is a previously discovered node, then the previous one is added to the parent's neighbor list. Eventually, all nodes will have been previously discovered, leaving OPEN empty. The algorithm then terminates, leaving a graph structure such as the one in Figure 7.2(c), "rooted" at the starting node. The most interesting feature of this structure is that all of the nodes contain in G_VALUE their distance from the starting node. This is what is meant by a graph "rooted" at the start, and is an extremely useful feature, as will be seen in the following sections.

In addition to building a graph rooted at START_STATE, GENERATE_GRAPH fills in a corresponding data structure which describes the features of the particular graph in question. This structure is a graph descriptor, and is shown in Figure 7.5. The DEPTH field tells the maximum distance of any node from the starting node. GENERATED tells how many nodes were created in the expansion of the graph. EXPANDED tells how many nodes were actually explored, and is thus a count of how many states there are in the graph.

Figure 7.5 Graph descriptor data structure.

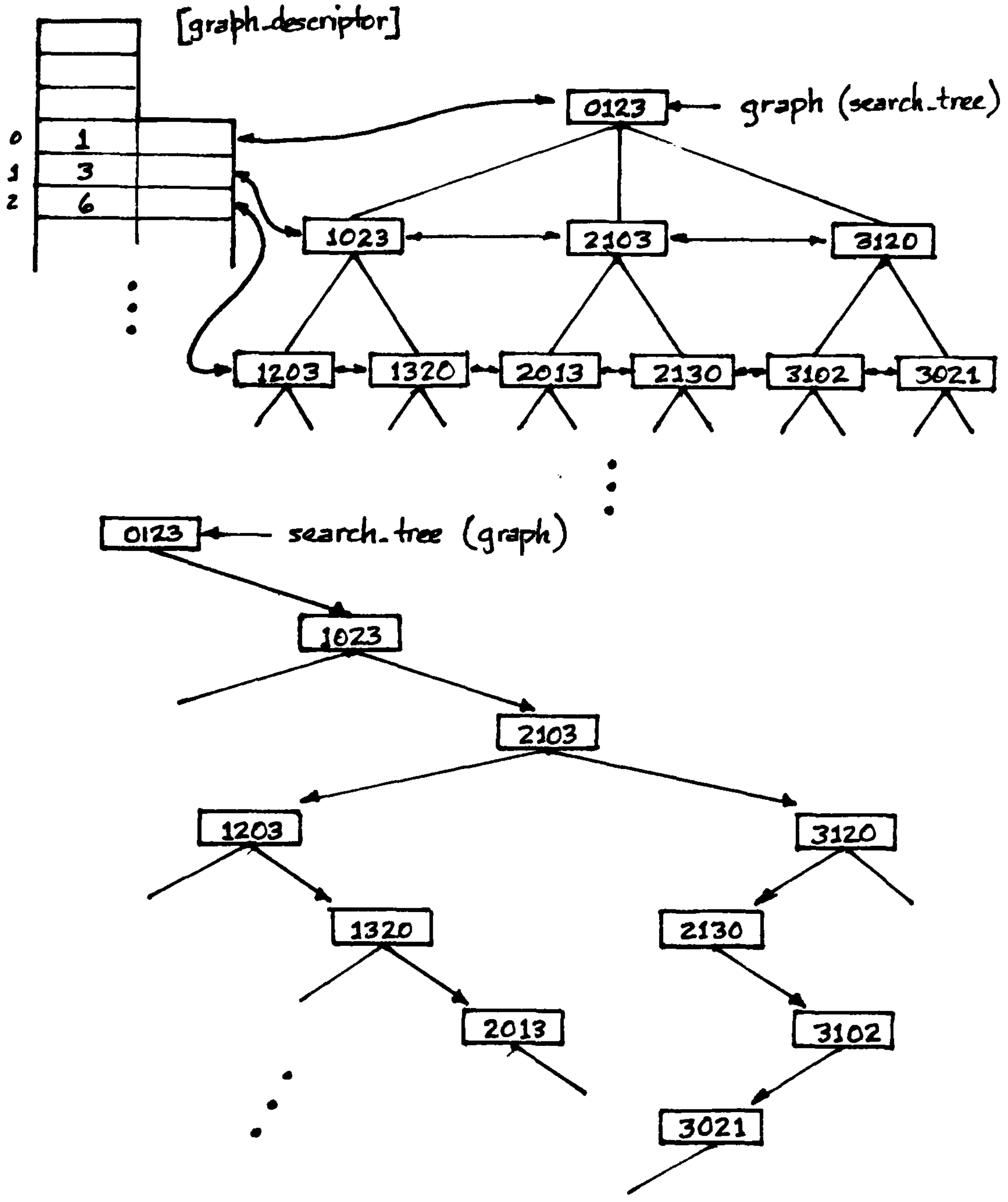


LEVEL is an array of records, each describing a certain "level" or group of nodes at the same depth in the graph. Each level record contains a count of the number of nodes at that level, and also contains a pointer to a doubly-linked list of these nodes. Space is provided in the array for up to 100 level records.

The procedure INITIALIZE_GRAPH_DESCRIPTOR sets all of the various counting fields to zero, and creates an empty level list for each of the levels in the level array. As GENERATE_GRAPH constructs a graph it keeps track of the number of nodes generated and expanded, and also the maximum depth. As each node in the graph is expanded, it is placed on the appropriate level list. The structures resulting from a call on GENERATE_GRAPH for a version of the "3-puzzle" are shown in Figure 7.6.

Before moving on to a discussion of the A* control structure implemented in this module, a description of the method of successor generation is appropriate. A general support procedure called GENERATE_SUCCESSORS accepts as input a puzzle state descriptor, and returns a list of newly created descendant puzzle nodes containing all of the states that may be obtained by one legal transformation on the input state. As described previously a legal transformation is defined as moving a bead to an adjacent, blank position that is connected by an arc or a link. GENERATE_SUCCESSORS is completely general to all Beads World configurations, and is used by the control structures

Figure 7.6 Data structures created by a call on GENERATE_GRAPH for the 3-puzzle. The binary search tree is shown separately for clarity, but is actually superimposed on the graph. Neighbor lists have been omitted for clarity.



implemented in this module. The three global Beads World configuration variables are contained in this module as static variables. NUM_POSITIONS tells how many positions there are in the puzzle. NUM_LINKS is an integer which tells how many outer positions are connected to the center with links. LINK is a boolean array with elements corresponding to each of the outer positions. If a link exists between a position and the center, then the corresponding boolean in LINK is TRUE.

The two versions of A* described in Section 5 above, ordered search and graph search, are both implemented in this module. ORDERED_SEARCH accepts as inputs a START and GOAL state, a HEURISTIC selector, and a WEIGHT, and returns a RESULTS description in the form of a results descriptor record. GRAPH_SEARCH has the same arguments. The basic A* algorithm and both the ordered search and graph search versions of it have already been described in general terms in previous sections of this document. Highlights of the implementations of these are described below.

Rather than maintain two distinct node lists, OPEN and CLOSED, these implementations maintain an OPEN list and a binary search tree. OPEN contains only those nodes which have been generated but which remain to be expanded, and is ordered by increasing F value. The search tree is a binary tree on which all generated nodes are placed, in order, alphabetically by their character state descriptions. With these structures, a binary tree search is performed to see if a node has been previously discovered. A node is defined to be CLOSED if it is in the tree but not on OPEN.

ORDERED_SEARCH does not maintain a graph structure among the nodes visited during the search -- it only maintains an implicit search tree of parents and successors. Thus, the neighbor list fields are not used and no neighbor lists are kept. When nodes are rediscovered on shorter paths, they are simply reexpanded. Backwards chaining parent pointers maintain the implicit search tree.

GRAPH_SEARCH also maintains an implicit search tree by keeping parent pointers among all successor nodes. However, this tree structure is superimposed on a graph structure which represents the subset of the complete Beads World graph that has been explored to that point in the search. As each node is generated, the binary search tree is searched for its state. If it is a new node, it is added to its parent's neighbor list, its parent pointer is directed to the parent node, its F value is calculated, and it is placed on OPEN and the binary search tree. If it is a rediscovered node on a shorter path, then the new node is discarded, and the old one is updated with the new path information. If this node was CLOSED, the results of the change in path information are propagated throughout the sub-graph by a recursive update procedure. If this node was OPEN, it is replaced on OPEN at the proper location for its new F value.

In order to provide a cleaner packaging of these routines, all of the information about a search run is packaged in a record structure called a RESULTS_DESCRIPTOR, which is passed as a VAR parameter and is filled in by the search routines. The RESULTS record contains a boolean, SOLVED, which indicates whether a solution path was found from the starting state to the goal state. It also has fields which hold the PATH_LENGTH, the number of nodes GENERATED, and the number of nodes EXPANDED. The minimum path length, MIN_PATH_LENGTH, is filled in by the calling routine, as it is usually provided with the start and goal states. The HEURISTIC and WEIGHT fields indicate which heuristic and weight were used in the solution of a particular puzzle. Finally, two pointer fields, START and GOAL, point to the starting node and the last node (goal node in the case of a successful solution) on the path. INITIALIZE_RESULTS simply clears the results descriptor. A utility function, PRINT_PUZZLE_SOLUTION, is provided; this routine accepts a results description record and prints the puzzle states contained in the nodes on the path from the start to the goal.

3. HEURISTICS

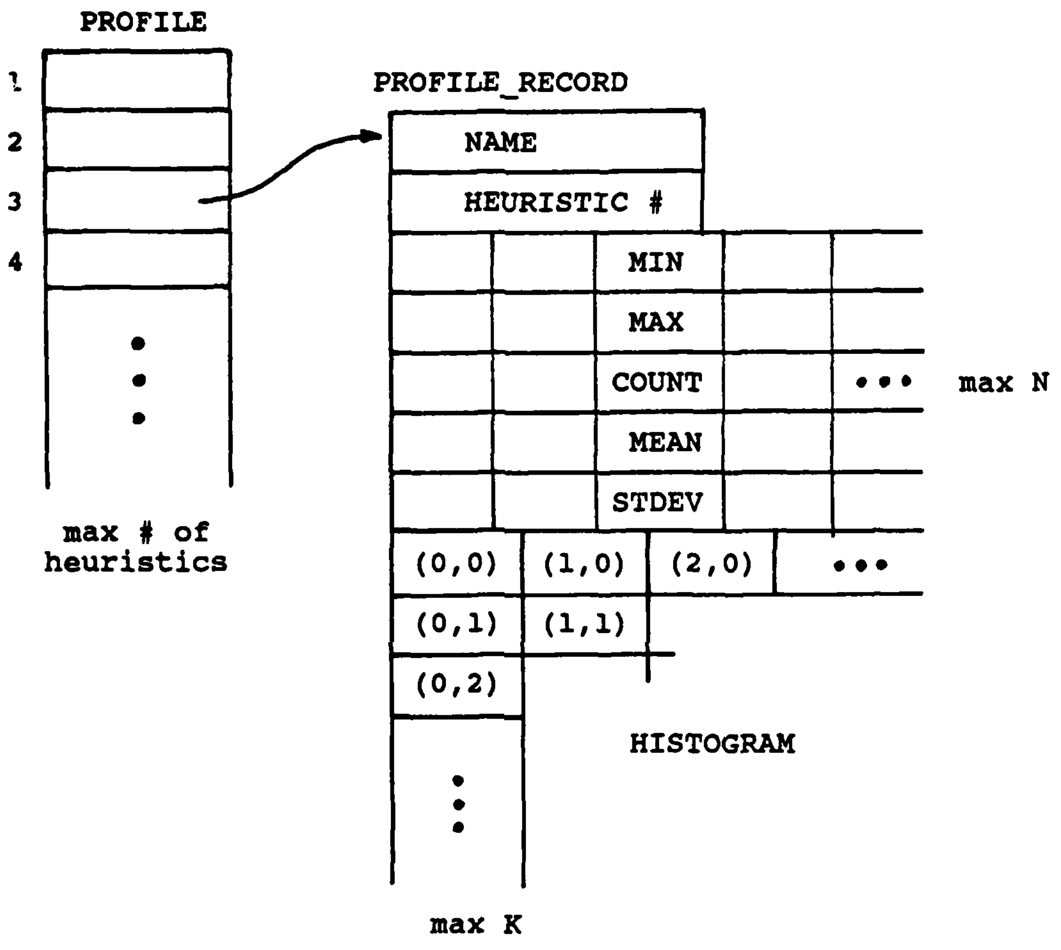
This module contains all of the functions used to calculate heuristic estimates of distance to the goal, and is intended as the module which future users will alter most frequently to suit their particular research needs. It contains routines which fall into three categories: those associated with abstracting the three traditional "8-puzzle" heuristics to the general Beads World, those which provide the ability to generate and use heuristic profiles, and those which provide other modules with access to these functions. Each of these categories are described below. Alteration and use of the heuristic module is described in a later section.

The basic heuristic module contains three routines which abstract the three traditional "8-puzzle" heuristics described in Section 6. TILES_MISPLACED accepts the current state and the goal state as inputs and returns a count of the number of tiles (beads) which are not in the same positions. MANHATTAN_DISTANCE abstracts the idea of shortest "city-block" distance into the idea of the smallest number of moves to put beads in their proper positions, assuming that there are no beads in the way requiring movement. ENHANCED_MANHATTAN_DISTANCE corresponds directly to the third heuristic, and counts the score of nodes out of sequence about the perimeter of the puzzle. These routines are, as with the GENERATE_SUCCESSORS procedure, completely generalized to all Beads World configurations, and are implemented with the support of three routines which calculate the minimum distances between states using moves constrained to go through center or perimeter positions.

An important and useful tool for analyzing the behavior of heuristic functions and for simulating this behavior at varying levels of abstraction is the ability to profile heuristic functions. This involves recording, for each call on a particular heuristic function (h), the true remaining distance to the goal (n) and that heuristic's estimate of the remaining distance (k). The resulting data is, for each heuristic, a set of tuples representing the (n, k) combinations encountered and the frequency of each combination. The heuristics module automatically collects this data in a large, three-dimensional, integer array with indices h, n, and k. This array is initialized to be all zero when the heuristics module is initialized. When all of the heuristic estimates have been performed, this data may then be written out to a special profile output file (in a format described later) by a call to the procedure PRINT_PROFILES, which expects a file name string as input.

Profiles are also useful as input to heuristic functions which attempt to simulate or model the behavior of some actual heuristics. Profile data such as that generated by previous search runs in the manner described above, or data contrived by the researcher, may be read in from auxiliary input files and stored in special record structures by use of the READ_PROFILES procedure. READ_PROFILES accepts a 30 character file name string as input, and reads the profile data from this file into a profile database which is maintained by the heuristics module. The structure of this is shown in Figure 7.7. PROFILE is a static array of profile pointers, one corresponding to each of the heuristics implemented in the module. Each of these pointers points to a PROFILE_RECORD. Profile records contain a 10 character NAME field, an integer HEURISTIC identification number, and six arrays. The first three, MIN, MAX, and COUNT, are arrays of integers, and record the minimum and maximum estimates at each level n, as well as the total number of estimates at each n. MEAN is a real array which records the mean estimate at each n. STDEV records the standard deviation about the mean at each n. The final array, HISTOGRAM, is a two-dimensional integer array which records the frequency of occurance of each (n, k) pair. These fields are calculated and filled in by READ_PROFILES as it processes the input file data. Only profiles for a few, particular heuristics are used at any one time. To save space, profile records are dynamically allocated and initialized by CREATE_PROFILE, which returns a pointer to an empty profile record. After being filled in, these profile records are entered in the PROFILE array. The data in these records is then available to applications by subscripting the PROFILE array with the heuristic identification number.

Figure 7.7 Profile database structures.



Heuristics are made available to external modules through the function ESTIMATED_DISTANCE, which accepts four input arguments. The first, H, is the heuristic selection number. CURRENT is a pointer to the node which is currently being examined. GOAL is the desired goal puzzle state. C_STAR is the minimum path cost from the start to the goal; this information is used only for worst case simulation and modeling. ESTIMATED_DISTANCE returns a natural number which gives the selected heuristic's estimate of the distance between the state of CURRENT and the GOAL. ESTIMATED_DISTANCE performs two functions. The first is simply to use the heuristic selector H to invoke the proper function from a CASE statement. The second function is more involved. For several heuristic functions, particularly those concerned with simulation and modeling of heuristics, it is necessary to know the true remaining distance to the goal, n. (This is of course also necessary for the collection of the profile data.) In order to obtain this value, ESTIMATED_DISTANCE invokes the GENERATE_GRAPH procedure from the control module to generate a graph rooted at the goal state. As was described in the previous section, GENERATE_GRAPH returns pointers to the graph and to a binary search tree superimposed on this graph. Each node in the graph contains as its G_VALUE its minimum distance from the root of the graph, or in this case, the goal. By doing a simple search for the current state in the search tree, the minimum distance n can be quickly obtained. Of course, the overhead of generating this graph is considerable, so the graph is saved and reused as long as the goal state remains the same. This method is significantly more efficient than using an admissible heuristic with A* to calculate this distance every time.

4. STATISTICS

Although not strictly a part of the code directly associated with A*, the statistics module provides routines which are useful in the simulation and modeling of heuristic functions, and is thus included in the set of tools provided. In addition, the applications which are described in this document rely heavily on the functions in the module.

STATISTIC provides functions which generate pseudo-random numbers, either evenly distributed within some range of values, or conforming to some desired distribution. The underlying function used is MTH$RANDOM, which is provided by a math and statistics library on the VAX, and which returns uniformly distributed random real numbers in the range [0,1], given an integer seed. RANDOM_INTEGER_BETWEEN uses this function to generate uniform random integers between the bounds provided.

The technique used to generate random numbers conforming to some non-uniform distribution is a little more complex, and requires some background theory. Remember that the only random generator available generates uniform random numbers in the range [0,1]. Suppose however that we wish to obtain randoms whose values occur with a frequency described by some density curve, such as the normal curve shown in Figure 7.8(a). There is another way of representing this desired distribution of values, using

Figure 7.8(a) Normal distribution density curve.

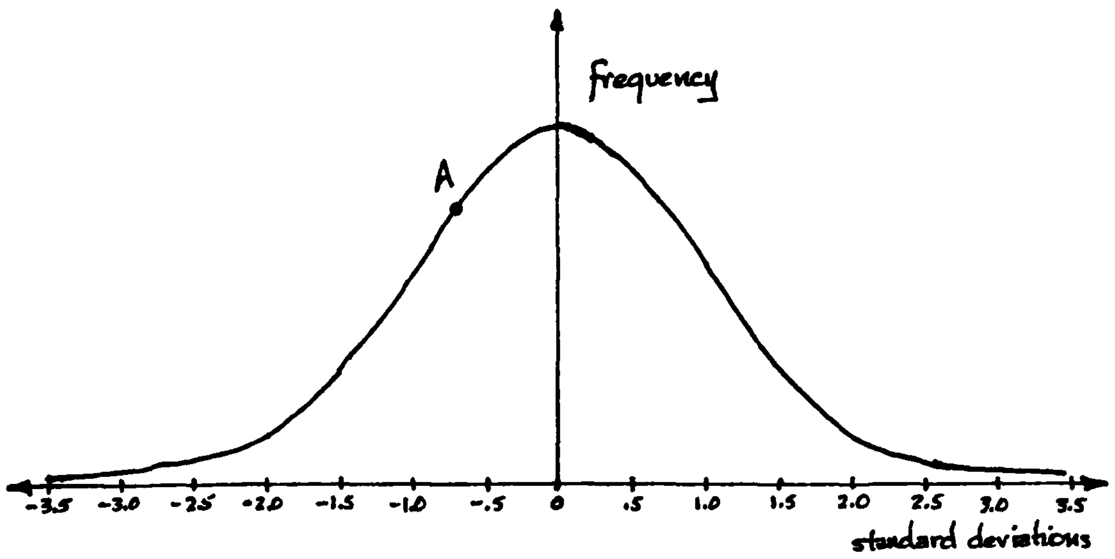
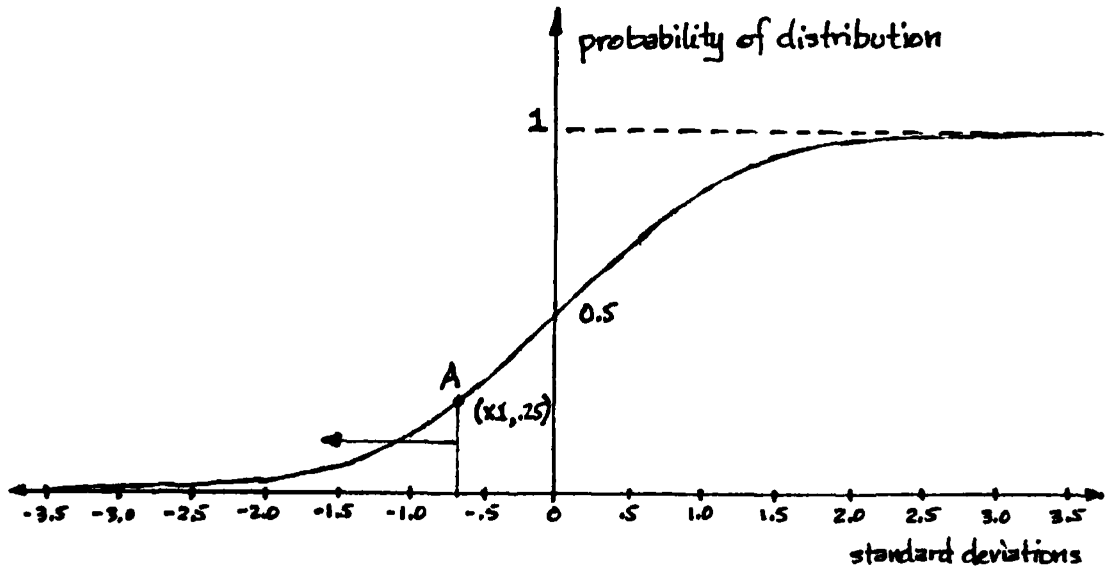


Figure 7.8(b) Distribution function for the density curve of Figure 7.8(a).



what is called the probability distribution function, shown for this example in Figure 7.8(b). This function is effectively the integral of the density curve, normalized to a value of 1, and represents the probability that the variable in question will fall below the domain value at that point. The other way to view this is as the summation of the area under the density curve, or the running total of the frequencies of the domain values. Thus, point A, with coordinates (X1,0.25), is the point at which there is a 25% probability that the X value will fall less than X1, or alternately, that 25% of the total number of X values will be less than X1.

The technique used to obtain randoms obeying this distribution is as follows. First a uniformly distributed random number lying in the range of the distribution function, [0,1], is obtained. The corresponding domain (X) value is then calculated or otherwise extracted, giving one "hit" at that value. In the aggregate, this process is essentially reversing the function that is the "area under the curve", or, in other words, taking the derivative of this distribution function. This yields X values whose frequencies conform to the desired density curve.

Another way of stating this (intuitively) might be as follows: 25% of the time the random number generator will return a number between 0 and 0.25. The inverse function will therefore return a number between - and X1 (refer to Figure 7.8) 25% of the time, as it should. If the 25% is replaced by some arbitrary percentage between 0 and 100, it can be seen that the generator is returning numbers in the proper range of values, exactly the correct percentage of the time. Therefore, it must be generating numbers with the correct frequency.

The most general way for the user to describe distribution functions is through enumeration of their domain and range values at sufficiently many discrete points. STATISTIC provides this capability through the use of distribution records. A DISTRIBUTION_RECORD contains a name field, which specifies the type of the distribution, and a field which tells how many pairs are enumerated. The third and fourth fields, ABSCISSA and ORDINATE, are real arrays which specify the function values at each of up to 100 discrete points. DISTRIBUTION_TYPE is an enumerated type which specifies each of the possible distributions available to importers of this module. Distributions are accessed through the DISTRIBUTION array, which contains pointers to all of the distributions currently available. These distributions may be created by reading them from an auxiliary file using the READ_DISTRIBUTIONS procedure provided.

Randoms conforming to a particular distribution are obtained by the real function RANDOM_BY_DISTRIBUTION, which accepts as input a constant of type DISTRIBUTION_TYPE specifying the desired distribution. Creation of distributions is discussed in the "use" section below.

D. USE OF THE BEADS WORLD TOOLS

The previous sections have presented in some detail the essential data structures and algorithms implemented in the four primary tools modules. Before going on to describe our applications, which use these tools, it will be useful to describe how to incorporate these tools into applications.

In order to use the data structures and procedures provided by a tools module, it is necessary to import this module. This involves including in the applications module all of the necessary data type, variable, and procedure declarations, with appropriate external references in the latter two cases, and then using the VAX Linker to link these compiled modules together. Because type checking is not performed across module boundaries, it is imperative that all declarations of data and procedures match exactly in every module. In order to make this convenient, special definition files for each tools module are included in this package.

The definition files are listed in Appendix B. As an example, the definition file corresponding to CONTROL.PAS is called CONTROL.DEF. It was created by deleting all code and all local procedures from CONTROL.PAS, leaving only the native data declarations and global procedure headings. By including this file in an application module and then deleting all unreferenced declarations, the user is able to provide all of the necessary linkage with the CONTROL module without worrying about data type or argument list agreement. Of course, this mechanism is only useful as long as the .PAS and .DEF files are in complete agreement. This means that whenever changes are made to an existing tools module, its corresponding definition file must be updated, as well as any other existing modules which import it. This mechanism is certainly not as convenient as the true module capability provided by languages such as Modula 2, but it is better than re-creating all of the definitions each time a new application is written.

The next important thing to discuss is the proper initialization of each of these modules. The controls module, heuristics module, and statistics module each have static data structures which need to be created and/or assigned initial values before the routines inside these modules are used. In order to make this initialization convenient, each module exports an initialization procedure which performs this when called. A good rule to follow when using these modules in applications is to call the initialization procedure for each module that is imported. Because the control module already imports from the heuristics module, which in turn imports from the statistics module, it turns out that applications that use these modules need only call the INITIALIZE_CONTROLS procedure exported by the controls module. Each of the initialization routines may be invoked separately, and multiple invocations cause no undesirable side effects.

The auxiliary data files which contain the profiles and distributions used by the heuristics and statistics modules are organized into specific formats. Profiles of each heuristic contain a header line which has a ten character name field, the heuristic number, and the number of entries for that heuristic. Each entry has four numbers. The first is the true distance n. The second is the estimated distance k. The fourth is the total number of times that this k was obtained for this n. The third field is the percentage out of all the samples at that n, effectively normalizing the histograms at each n. Several profiles can be included in the same file. Figure 7.9 illustrates the file format of profile data.

Figure 7.9 Example of the data in a profile auxiliary file.

```
profile name       heuristic #      number of entries

              kl         1          72
              0     0   100.0      100
              1     1   100.0      253
              2     2   100.0      402
              3     3   100.0      670
              4     3    30.5      328
              4     4    69.5      748
              5     3    19.8      326
              5     4    29.9      493
              5     5    50.3      828
              6     3    16.6      421
              6     4    19.4      492
              6     5    51.2     1297
              6     6    12.7      322
              7     4    25.1      928
              7     5    52.5     1939
              7     6    22.4      829
              8     3     5.9      296
              8     4    11.5      573
              8     5    56.8     2841
              8     6    25.8     1293
              .     .     .          .
              .     .     .          .
              .     .     .          .

             [n]   [k]   [%]   [frequency]
```

Figure 7.10 shows the format of a distribution file. The first line of each distribution is also a header. The first field is a constant of the enumerated type DISTRIBUTION_TYPE, and identifies the distribution. The second item is a number telling how many discrete point pairs follow. Each line after that consists of two real numbers, the abscissa and ordinate, describing the distribution function at each discrete point. Several distributions may also be included in the same file.

Figure 7.10 Distribution file format.

```
     type      number of entries
        \         /
         \       /
        normal      57
        -4.0       0.0000
        -3.5       0.0002
        -3.0       0.0013
        -2.5       0.0062
                .
                .
                .
        -0.5       0.3085
        -0.4       0.3446
        -0.3       0.3821
        -0.2       0.4207
        -0.1       0.4602
         0.0       0.5000
         0.1       0.5398
         0.2       0.5793
                .
                .
                .
         3.5       0.9998
         4.0       1.0000
          ↑          ↑
     [abscissa]  [ordinate]
```

The two additions or changes that users writing new applications with these tools are most likely to make are the addition of heuristics and distributions to those already provided. Heuristics may be easily added to the heuristics module. First, the heuristic function is created and inserted. Then an identification number is assigned to this heuristic, and a corresponding call to the function is added to the CASE statement in

ESTIMATED_DISTANCE. The system currently allows for up to 24 heuristics, but this number can be easily increased.

The addition of different distributions is also fairly simple. A name for each new distribution must be added to the DISTRIBUTION_TYPE list; these names are the key to accessing distribution functions from the heuristics module. Then the corresponding distribution function data must be created in the proper file format, calculated either by hand or by a throw-away program.

E. APPLICATIONS OF THE BEADS WORLD TOOLS

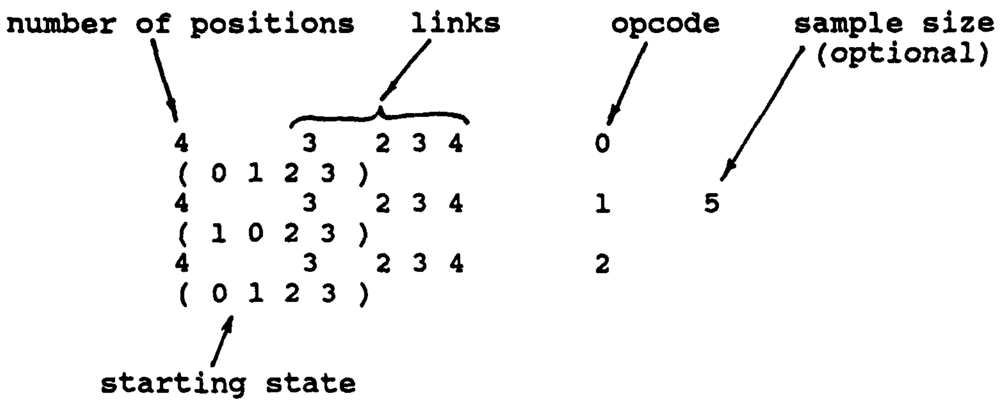
Three primary applications programs were developed to generate the data that is presented in the other sections of this document, and to display that data in meaningful form. Two of these programs, the ones responsible for gathering the data, incorporate the tools that have been described in the previous pages. The third program is a FORTRAN graphics package that displays the data in a variety of formats, and does not use any of the tools previously described. These three applications are described in the following sections, which serve as both the sole documentation for these programs and as examples of the use of the tools previously described. Future users of this software package may be able to use these applications directly, or may wish to alter them or use them as models to create programs more tailored to their specific needs.

1. GRAPH GENERATION AND ANALYSIS

The first application described in this section is the GRAPH_SPACE program module listed in Appendix C. This program provides the user with three capabilities. By varying the commands and data in the input file, the user can generate the entire graph for a given Beads World configuration and list its features; the user can generate a sample set of start and goal state pairs from the graph for use as input to search algorithms; finally, the user can actually print all of the puzzle states in the graph.

A sample input data file for GRAPH_SPACE is shown in Figure 7.11. Each operation is represented by two lines of data. The first line specifies the configuration of the Beads World to be used, as well as the operation to perform. The first two items in the line are the number of positions and the number of links, in that order. Following that are the position numbers of the links. The next number is the opcode. There are three possible opcodes: 0 causes generation of the graph and the printing of its characteristics; 1 generates sample state pairs from the graph; 2 causes the graph to be printed. If the opcode is a 1, then an optional last field specifies the minimum number of sample states to draw from every level of the graph.

Figure 7.11 Sample input data for GRAPH_SPACE. Note that three operations are specified by the file.



```
4     3   2 3 4      0
( 0 1 2 3 )
4     3   2 3 4      1     5
( 1 0 2 3 )
4     3   2 3 4      2
( 0 1 2 3 )
```

The second line of input data contains only one item; this is the starting puzzle state at which the graph is to be rooted.

After reading in the two lines which form each command, GRAPH_SPACE invokes the GENERATE_GRAPH procedure from the controls module. This returns a pointer to the graph rooted at the starting state, a pointer to the binary search tree containing the nodes of the graph, and a graph description record. If the opcode is a zero, then an output procedure prints the puzzle configuration, the starting state, and the information contained in the graph descriptor. As a visual aid this procedure also generates a histogram of the number of nodes at each level in the graph. An example of this output is shown in the tables in Section 3.

If the opcode is a 1, then GRAPH_SPACE generates sample (start, goal) puzzle state pairs from the graph. This is done by repeatedly extracting at random a puzzle state from each level of the graph. If a minimum sample size is specified, then at least that many states are taken from each level, if available. In addition, in order to reflect the distribution of states in the graph, one-hundred additional states are extracted, in approximate proportion to the graph space histogram mentioned above. These states are printed out in pair with the root state of the graph. In addition, the distance between these states, which is the level from which each goal state was extracted, is also printed.

The third option (opcode = 2) causes GRAPH_SPACE to enumerate the states in the graph. This is done recursively by neighbors until the nodes at the last level are reached. An example of this is shown for the 3-puzzle in Figure 7.12. At some 60 lines per page, this procedure could be quite expensive for puzzles larger than the 6-puzzle.

Figure 7.12 Puzzle states for the 3-puzzle.

```
( 0  1  2  3 )
( 3  1  2  0 )
( 3  1  0  2 )
( 0  1  3  2 )
( 2  1  3  0 )
( 2  1  0  3 )
( 2  0  1  3 )
( 0  2  1  3 )
( 3  2  1  0 )
( 3  2  0  1 )
( 0  2  3  1 )
( 1  2  3  0 )
( 1  2  0  3 )
( 1  0  2  3 )
( 1  3  2  0 )
( 0  3  2  1 )
( 2  3  0  1 )
( 2  3  1  0 )
( 0  3  1  2 )
( 1  3  0  2 )
( 1  0  3  2 )
( 3  0  1  2 )
( 2  0  3  1 )
( 3  0  2  1 )
```

2. PUZZLE SOLUTIONS WITH A*

The second application program, SOLVE, is listed in Appendix C. This program provides a very flexible vehicle for gathering data on A* search. By varying the input commands and data provided to SOLVE, the user can run either the ordered search or graph search versions of A*, using any heuristic, at any weight, for any of the possible Beads World configurations. In addition, the user has a variety of options for collecting, synthesizing, and displaying this data.

Figure 7.13 Sample input data for SOLVE.

```
7      3  2 4 6          3    ordered
3   1 2 3             4  0.2 0.5 0.7 1.0
prof_in = "  "     prof_out = "  "
distribution_in = "  "
8 ( 0 1 2 3 4 5 6 )  ( 2 5 0 1 3 4 6 )
8 ( 0 1 2 3 4 5 6 )  ( 2 5 1 3 0 4 6 )
8 ( 0 1 2 3 4 5 6 )  ( 0 4 2 5 3 6 1 )
8 ( 0 1 2 3 4 5 6 )  ( 4 5 1 2 3 6 0 )
8 ( 0 1 2 3 4 5 6 )  ( 4 5 1 2 0 3 6 )
8 ( 0 1 2 3 4 5 6 )  ( 5 6 1 2 0 3 4 )
8 ( 0 1 2 3 4 5 6 )  ( 4 2 0 5 3 6 1 )
8 ( 0 1 2 3 4 5 6 )  ( 0 6 1 2 3 5 4 )
```

The format for the input data is shown in Figure 7.13. As with the GRAPH_SPACE program described above, the first line of input configures the programs for the proper Beads World model. Opcodes can assume values of 0 to 3; each of these resulting operations are described below. Finally, the last item on the first line specifies which search method is to be used: ordered search or graph search.

The second line of input tells SOLVE which heuristics and which weights are to be applied to each problem. The first item is the number of heuristics. This is followed by a list of the numeric identifiers of those heuristics. The second set of numbers is the number of desired weights, followed by a list of those weights.

The third line contains two items, both of which are optional. The first item is the file name of the profile input file, enclosed in '"' delimiters. If no profiles are needed, then this field may be left blank (i.e. no characters between the delimiters). The second item is the profile output file name. If this is not specified then no output files are created.

The fourth line also contains an optional file name field, which specifies in what file the distribution function description data is located. If not specified, no attempt is made to create distribution records, and heuristics which use distributions cannot be used.

The input data consists of an unspecified number of puzzle problem entries. Each entry consists of three values. The first is the minimum distance between the start and goal states in the graph. The second and third items are the state descriptions of the start and goal, respectively. Note that these entries are in the same format as those generated by GRAPH_SPACE when instructed to generate sample pairs.

As mentioned before, the opcode can specify one of four distinct actions. The first (opcode = 0) results in the printing of a complete description of the puzzle problem and of the performance of the search algorithm in finding a solution. The second option (opcode = 1) is the same as the first, with the additional feature of printing the puzzle states lying on the solution path. The third option (opcode = 2) is used whenever the data set is too large to be displayed using one of the first two options, or when it is more informative to see the data condensed for comparison. The raw data is simply printed for each puzzle instance (start and goal pair at every heuristic and weight) according to a specific format. Figures 7.14(a) through 7.14(c) give examples of each of these options.

The fourth option is significantly different from the first three. In order to provide an aggregate measure of the performance of the algorithm on the input data across the heuristics and weights, the results of each run must be aggregated for each level n. The format of this performance data is shown in Figure 7.14(d).

There are two basic measures of performance. The first is the number of nodes expanded. For every n and every heuristic at every weight, there are three entries: the minimum number of nodes expanded, the maximum number of nodes expanded, and the mean number of nodes expanded. The second performance measure is path length, and again for every n, heuristic, and weight, there are three entries: minimum path length, maximum path length, and mean path length. This data is organized into two groups of three lines each, as shown in Figure 7.14(d). This data is also in the proper format for processing by the graphical display package which is described in the next section.

Figure 7.14(a) Printed results using option '0' with SOLVE.

```
  Positions :   7  Links :  2  4  6
  Heuristics :   1  2
  Weights :    1   0.70

PROBLEM SOLUTION RESULTS   2  0.70

  Start:  ( 2  5  0  1  3  4  6 )
  Goal:   ( 0  1  2  3  4  5  6 )

  Nodes Generated     :    22
  Nodes Expanded      :     8
  Path Length         :     8
  Minimum Path Length :     8
```

Figure 7.14(b) Printed results for option '1' with SOLVE.

```
  Positions :   7  Links :  2  4  6
  Heuristics :   1  2
  Weights :   1  0.70

PROBLEM SOLUTION RESULTS   2  0.70

  Start :  ( 2  5  0  1  3  4  6 )
  Goal :   ( 0  1  2  3  4  5  6 )

  Nodes Generated     :     22
  Nodes Expanded      :      8
  Path Length         :      8
  Minimum Path Length :      8


( 2  5  0  1  3  4  6 )
( 2  5  1  0  3  4  6 )
( 0  5  1  2  3  4  6 )
( 5  0  1  2  3  4  6 )
( 5  1  0  2  3  4  6 )
( 5  1  2  0  3  4  6 )
( 5  1  2  3  0  4  6 )
( 5  1  2  3  4  0  6 )
( 0  1  2  3  4  5  6 )
```

Figure 7.14(c) Printed results for option '2' with SOLVE. Note the tabular form.

```
   7          3   2   4   6
   3   1  2   3           4   0.20   0.50   0.70   1.00
   1    0.20      8       8    186     72
   1    0.50      8       8     39     14
   1    0.70      8       8     57     21
   1    1.00      8      58   9964   3791
   2    0.20      8       8    131     50
   2    0.50      8       8     22      8
   2    0.70      8       8     22      8
   2    1.00      8       8     22      8
   3    0.20      8       8     30     11
   3    0.50      8       8     24      9
   3    0.70      8       8     24      9
   3    1.00      8       8     24      9

heuristic  weight                   generated   expanded
         min. path length  path length
```

Figure 7.14(d) Printed aggregate results for option '3' with SOLVE.

```
   7          3    2    4    6
   3     1 2 3               4   0.20   0.50   0.70   1.00
   8     1    70   14   10       8
   8     1    75   18   22    2126
   8     1    84   22   34    3827
   8     1     8    8    8       8
   8     1     8    8    8      47
   8     1     8    8    8      60
 ------------------------------------------------------
   8     2    47    8    8       8          kmin
   8     2    55    9   10     191          kmean
   8     2    66   16   22     915          kmax
   8     2     8    8    8       8          lmin
   8     2     8    8    8      10          lmean
   8     2     8    8    8      18          lmax
 ------------------------------------------------------
   8     3    11    8    8       8
   8     3    13   12   12      21
   8     3    20   17   19      39
   8     3     8    8    8       8
   8     3     8    8    8      13
   8     3     8    8    8      20

N     heuristic      (one entry for each weight)
```

3. GRAPHIC DISPLAY OF RESULTS

The SOLVE application described above generates large volumes of data which, if left in tabular form, can be rather difficult to analyze and evaluate. Transforming such tables into graphical form makes it easier to detect trends and to observe interesting behavior in the data. A good picture is worth a thousand words, and in this setting, graphs serve to provide a descriptive and precise summary of the execution results.

A powerful commercial graphics package called DISSPLA is available on the RICC VAX-11/780; it was found to be a versatile, well-documented, and fairly easy-to-use package that not only offered all of the graphing formats required, but also permitted their review on either the terminal or in hard copy. However, there is such a variety of methods and combinations in which to view the volumes of data that it became necessary to create a tool to gather the specific graph parameters from the user, extract the required data from the data base, and make the necessary calls to DISSPLA to finally provide the graph. This tool takes the form of a basic program framework, rather than a general, all-encompassing package, because it is tied so closely to the data, and because of the varied nature of the graphs required. This tool provides a basic, moldable framework that can be easily tailored to specific needs.

The basic framework is written in VAX FORTRAN, and consists of code performing four distinct tasks: (1) input of data, (2) menu operation, (3) axis and graph set-up, and (4) curve plotting. The input routine reads the data from the appropriate aggregate data output file generated by SOLVE. The menu presents the user with a variety of viewing options and plotting combinations from which to select. The axis set-up routines establish the appropriate type of graph with DISSPLA, based on the parameters selected by the user in the menu routine. Finally, the curve plotting routine extracts the appropriate data from the data base for each curve selected by the user, and calls DISSPLA routines to plot and label each.

Appendix C includes the code from two of the main graphing tools created from this basic framework: GRAFER and GRAFPROF. GRAFER generates four types of complexity graphs: (1) X versus N (see Figure 6.1(b)), (2) X versus W (see Figure 6.9), (3) L versus N (see Figure 6.6), and (4) L versus W (see Figure 6.15). Input to GRAFER must be in the format shown in Figure 7.14(d), residing in a file named SEARCH.OUT.

The other routine, GRAFPROF, provides profile graphs in one of two forms: (1) two-dimensional K versus I (see Figure 8.2(a)), and (2) three-dimensional K versus I versus Frequency (see Figure 8.2(b)). The input to GRAFPROF must be in the format depicted in Figure 7.9, residing in a file called PROFILE.RUN. (Source profiles are optionally read from a file called PROFILE.PRO.)

As indicated, these graphing routines stem from the same general framework. Other minor modifications resulted in variations that created the graphs shown in Figures 5.6 through 5.17 (directly comparing Graph search and Ordered search), and Figures 5.18 through 5.26 (comparing F and G as discriminators in A*). These are mentioned as testimony to the versatility of the framework, should future research require modification of these applications.

F. ADDITIONS AND ENHANCEMENTS

As with any large and complicated piece of software, this package has undergone almost constant evolution, as use has brought out shortcomings and possible improvements in its features. In fact, the authors do not view this software as complete, but rather as evolved to a stable enough point to be useful to other researchers. In this spirit, several possible additions and enhancements are suggested, to make this package an even better tool. These suggestions fall into two categories, discussed below.

Most of the suggestions for improvement fall under the heading of efficiency. It was felt to be very important to maintain the flexibility, maintainability, and understandability of this package. To a large extent, careful packaging and the use of modules has accomplished this. However, this has also meant that some sacrifices have been made in efficiency. The major limitation on the use of this package is the tremendous amount of CPU time that is required to obtain a statistically significant amount of data. One of the largest time costs associated with this implementation is the large number of system calls performed while doing dynamic allocation. A major savings could be obtained by replacing the dynamically allocated neighbor lists with static neighbor arrays in each puzzle node. Ultimately, this does not greatly increase the required memory, and does significantly reduce the number of system calls. Even more drastically, if the user is willing to restrict the maximum beads world puzzle size to, for example, the "6-puzzle", then static allocation is feasible for all of the puzzle nodes as well. Dramatic time savings would then be possible.

Other areas of efficiency include reducing the number of procedure calls and optimizing certain algorithms and sections of code. This usually involves obscuring the functionality of the code, but may be worthwhile if sufficient CPU time savings are realized.

The second area of improvement is that of "user-friendly" operation. The primary motivation for using input data in the standard input file as both command and data was to allow the use of the batch queue to submit large data runs without tying up a terminal or account for several hours or days. However, it would be nice if some form of interactive or menu-driven command mode were provided to the user. This may involve playing complicated games with the DCL command language, and thus is left for others to work out.

VIII. SIMULATING HEURISTIC BEHAVIOR

A. INTRODUCTION

This chapter deals with a means of comparing the results of classes of heuristics whose statistical estimating behavior is stochastically the same. There are two related goals: (1) to empirically verify claims in the literature that heuristics sharing the same KMIN and KMAX estimate-bounding functions (called profiles) are "equivalent", and (2) to study how completely these statistical profiles capture the essence and power of the heuristic which they represent. This chapter proceeds by describing exactly what a profile consists of and how these profiles were constructed and used, followed by a discussion of the above objectives in detail, a summary of the simulated heuristics used, and finally concludes by examining the results obtained by simulation with these profiles.

1. WHAT IS A PROFILE?

In general, at any given time, there are many nodes in the search graph at distance i from the goal. The estimates calculated by a heuristic function (K) from the set of nodes at level i to the goal will span a range of values, with upper and lower limits which shall be called

KMIN(i) and KMAX(i), respectively. This set of numeric data collected over all i characterizes a heuristic in terms of the bounds on its error behavior. In addition to KMIN(i) and KMAX(i), other data that could be collected to further characterize a heuristic includes the mean of its estimates at each i (called KMEAN(i)), the standard deviation at each i (called ST_DEV(i)), and the frequency of each of the estimates for each i (called the actual distribution). This set of statistical data collected on a heuristic is referred to as a "profile".

2. EQUIVALENCE OF HEURISTICS

Gaschnig (1979, Pg 84) claimed:

> "Two K functions are equivalent iff their corresponding KMIN and KMAX functions are identical. We have blurred the distinction between all K functions that happen to have a particular KMIN and KMAX as bounding functions".

Therefore, if two heuristics are different in their manner of estimating the distance to the goal, but always arrive at some distribution of values within the bounds KMIN and KMAX, according to Gaschnig, they are termed 'equivalent'. Equivalent heuristics should produce similar results in terms of the number of nodes expanded, solution path length, and behavior at various weights. However, Gaschnig's 'definition' of equivalence appears to be somewhat simplistic, because even though two heuristics may share the same bounding functions, at any given node they can calculate entirely different results. (This is referred to as "timing".)

We indicated earlier that heuristics can be characterized by profiles that consist of the statistics KMIN, KMEAN, KMAX, ST_DEV, and the frequency distribution of values. If two heuristics were to share an entire profile, and not just KMIN and KMAX, they could be declared equivalent with greater confidence because the basis of the equivalence stems from a more specific description of their respective behavior. Although these additional measures don't guarantee that the two will behave identically at every node, at least they do insure that the two will have the same aggregate statistical behavior, which is not the case using only KMIN and KMAX.

One objective then, is to empirically study the results of several different heuristics whose 'equivalence' is based on various aspects of their respective profiles (i.e. heuristics bounded by the same KMIN and KMAX, or heuristics sharing the same KMEAN, or even heuristics sharing an entire profile).

3. COMPLETENESS OF PROFILE

Another way of looking at equivalence is to say that a full profile characterizes the heuristic completely enough that the profile could be used in place of the actual heuristic (in other words, the profile could be used to simulate the original heuristic), and the resulting performance in terms of number of nodes expanded should be identical. An additional objective then, is to see how completely the profile characterizes the actual heuristic on which it was based.

4. WHAT IS SIMULATION?

Typically, heuristics take the form of an equation or formula that evaluates particular aspects of a given node's state and calculates an estimate based on what it finds. For example, the heuristic K1 described in Chapter VI counted the number of tiles out of place as its estimate. In simulation, the basic technique is to eliminate the need for a formula or equation that is dependant upon a given node's configuration, introducing a level of abstraction between the heuristic and the domain by using statistics stemming from observed behavior elsewhere in the real world. The simulation is accomplished through use of profiles and "contrived" heuristics, where the contrived heuristic is a black-box that bases its estimate on the information given in another heuristic's profile rather than by calculating it from bead configurations. In essence, the contrived heuristic is a "copy-cat" heuristic that "says what the other guy said" and has no information of its own to offer.

Our contrived heuristics use the profiles simply as a 'look-up' device: to evaluate a given node, instead of looking at its bead configuration (which a real heuristic would have to do), the contrived heuristic looks up what another heuristic calculated for all nodes at distance i from the goal, and returns some value based on the statistics found in that profile. Since the profile is made up of several categories of figures, this makes available a variety of numbers to choose from and return. Some of the many values that could be returned include:

(1) KMIN(i)
(2) KMAX(i)
(3) KMEAN(i)
(4) KMEAN(i) +/- N standard devations
(5) a random value between KMIN(i) and KMAX(i)
(6) a random value normally distributed around KMEAN(i).
(7) a random value selected according to the actual distribution of estimates in the profile.
(8) KMEAN(i) + N standard deviations if on path, KMEAN(i) - N standard deviations if off path.
etc.

So essentially, a profile supplies the statistical performance information gathered on an actual heuristic, and the contrived heuristic decides which categories from that profile to use in forming its plagiarized estimate. This choice can have a dramatic effect on the performance results: consider the difference in search performance of a contrived heuristic returning KMEAN as its estimate, and another using KMAX. (One would expect better-than-average performance using the former, and worse-than-average results using the latter). However, since the typical real-world heuristic does not merely return one single value, to be more realistic, the contrived heuristic could emulate this behavior by varying its estimates using any of the techniques offered in options 5, 6, and 7 above.

The technique that the contrived heuristic uses to derive its plagiarized estimate from the profile ultimately defines a particular pattern or distribution of values. For example, if only KMAX is returned, the distribution of values returned over the course of the run will be the value KMAX, with no deviation. Using an option such as expressed in items 5, 6, or 7 above results in a wider distribution that spans several values at each i. The selection of which distribution to use depends upon the objective of the simulation: one could choose a distribution to focus on a particular type of behavior (like options 1, 2, 3, 4, and 8 above), or the objective might be to attempt to duplicate the performance of the original heuristic. The options and distributions that we implemented for our contrived heuristics are discussed in detail following a detailed description of the process used to build profiles.

B. GENERATING PROFILES

The simulation process depends upon having a good set of profiles derived from actual heuristics, and had to be gathered before any experimental work could proceed. This process involves collecting two figures on a given pair of puzzle configurations: one figure is the true minimum distance (i) between the two states, and the other is the heuristic's estimate of what this actual distance was.

The collection of these two values over a large number of state pairs (we call them start/goal pairs) characterizes the heuristic being used so that at any particular distance i, this profile could divulge the absolute minimum and maximum values observed, and also the mean and standard deviation of the aggregate sample.

Gaschnig (1979, pp. 39-42) used two methods to obtain figures for his profiles. For the first method, he used the 895 start/goal states of his solution sample as one set on which to gather figures. He knew the actual distances, which were determined as the pairs were created, and only had to let his heuristics estimate these distances to provide the necessary figures to build the profile. However, the results from this method were rejected because the sample size was based on only 895 values, which was felt to be too small to be meaningful.

The second method employed by Gaschnig was to use the nodes generated in the search tree during the solution of the 895 problems; by comparing each node generated with the root node, a greater number of pairs could be sampled. Since the actual distance of each node from the root was known (simply the value G since he used A* with an admissible F), all he needed to do was to let his heuristics provide their estimate of this distance, and collect the resulting values. This method provided 11,448 "start/goal" state comparisons.

Neither method appealed to us. The first method is based on too small a sample; the second method can bias the results, because it only samples nodes on or close to the solution path. Thus it might not represent the heuristic's full range of estimates.

An alternative method was devised instead. From a given start state, the complete state space was built by using the same basic process that generated the 198 start/goal pairs (described in Chapter IV), but modified specifically to gather the profile information that we needed. The random selection mechanism from that process was then employed to select a proportional number of nodes from each level of the state space. For each node selected, the heuristic estimated its distance from the start, or root node. The actual distance was the level of the node, or the value G, since the state space was built using A* with an admissible F.

This method was attractive because it randomly selected nodes at each level in proportion to the total number of nodes at each level of the state space. Also, the nodes selected were not on any particular path (unlike Gaschnig's second technique which tended to sample nodes along the route to the goal). So, not only were duplicates minimized, but in addition any node in the tree had an equal chance of being picked. Taken over a wide number of search trees, this method thoroughly tested the heuristics, and should provide profiles that represent the true range of values in a general setting.

1. COMPARING GENERATION METHODS

For comparison, we created profiles using both Gaschnig's second technique (referred to as Method A) and the alternative method described above (called Method B). Method A gathered heuristic estimates for K1, K2, and K3 during the solution of 198 start/goal pairs at seven weights, and Method B was created from 100 arbitrarily selected start configurations (and hence, 100 different search trees).

Table 8.1 compares the number of samples taken from the various levels using the two methods.

TABLE 8.1
Profile Sample Sizes

Level	Method A	Method B
1	1476	253
2	4538	391
3	3601	487
4	6252	478
5	7836	583
6	10004	578
7	16233	620
8	25064	683
9	49972	824
10	73930	1027
11	121274	1235
12	154682	1557
13	190686	1810
14	175902	1972
15	141106	1818
16	90395	1574
17	46899	1128
18	19614	763
19	8955	579
20	2938	396
Total Sample Size:	1,151,457	18,756

Table 8.1 shows that many more values were gathered using Method A (over 1 million compared to 18 thousand for Method B). However, Table 8.2 (below) compares the results from both profiling methods, and shows that not only is the span of values narrower using Method A (at level 8, Method A only had estimates 4, 5, and 6, while Method B included 3, 4, 5, and 6), but also that the distribution is different from Method B's values (at level 4, Method A returned the value 4 40% of the time, while Method B returned a 4 69% of the time), in spite of the fact that Method B had a much smaller number of values in its sample. Only two levels are shown from the values collected for

heuristic K1, but the results at all other levels and heuristics reflect this same trend.

TABLE 8.2
Disparity of Distributions

Actual Distance (i)	Heuristic Estimate (K)	Method A	Method B
4	3	60%	31%
4	4	40%	69%
.	.	.	.
.	.	.	.
.	.	.	.
8	3	0%	5%
8	4	32%	13%
8	5	54%	57%
8	6	14%	25%

Figure 8.1 provides a graphical representation comparing the values gathered using Method B and Gaschnig's profile for K3, and brings up an interesting point. In section 6, we were disturbed that the profile from our version of K3 was not the same as Gaschnig's because our graphs differed slightly (see Figure 6.21). However, the new profile appears to be in much greater agreement (Figure 8.1b). Note that our K3 (KMIN) underestimates when the distance from the goal (i) is high. However, observe that Gaschnig's K3 overestimated, again raising the question if our K3 was interpreted correctly. In spite of this apparent discrepency, we still claim our version of K3 is

correct, and for support, provide an 8-Puzzle example where the heuristic underestimates the true distance from the goal:

Start

6	7	8
	5	1
4	3	2

Goal

2	3	4
1	5	
8	7	6

$$K3 = K2 + 3 * seq$$

The actual distance to the goal in this example is 28 moves, or simply the rotation of each of the seven perimeter tiles four positions clockwise. Since the perimeter tiles are in their relative positions with respect to their neighboring tiles, SEQ is 0. The Manhattan Distance calculates the shortest route to the tile's goal position, which is four moves for tiles 2, 4, 6, and 8. Tiles 1, 3, and 5 need only two moves each, using the center of the puzzle as a shortcut. Hence, the Manhattan Distance becomes 22, and since SEQ is 0, K3 returns the value 22. This value is contrary to the behavior depicted at this distance for Gaschnig's K3, but is consistent with the results observed at large i from our K3 using Method B (Figure 8.1).

AD-A173 096 AN EMPIRICAL STUDY IN THE SIMULATION OF HEURISTIC ERROR BEHAVIOR(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH S R HANSEN 1986 AFIT/CI/NR-86-184T 3/4

UNCLASSIFIED F/G 9/2 NL

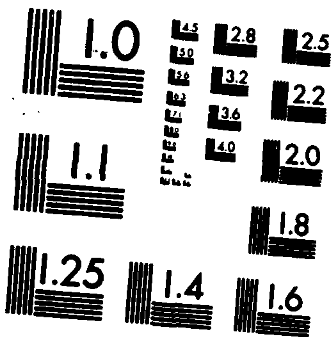

1.0
4.5
2.8
2.5
5.0
5.6
3.2
2.2
6.3
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

Figure 8.1
Profiles for K3

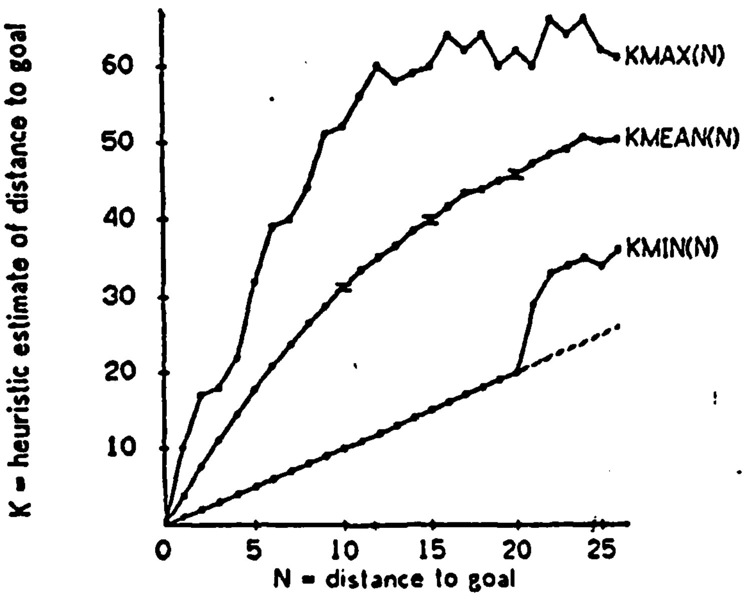


Figure 8.1a 8-Puzzle Profile

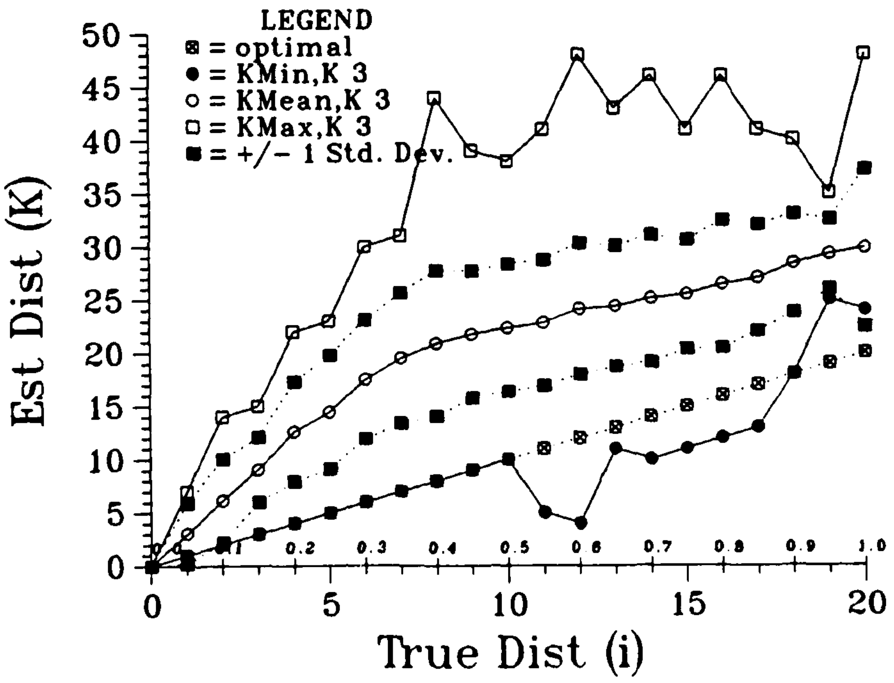


Figure 8.1b 6-Puzzle Profile using Method B

2. PROFILING CONCLUSIONS

There are 6 million possible combinations of start/goal pairs for the 6-Puzzle, and even though Method A sampled one million pairs, there was a tremendous amount of duplication in the samples selected since they were all picked from paths en route to a goal. Method B was based on fewer comparisons, picking 18,756 out of the 6 million for a selection ratio of 1 in 300, but was derived at random from 100 different search trees, giving the sample the potential for more breadth and hence, more opportunity to find the underestimating cases we observed in Figure 8.1. Also, Method B was not as prone to sample duplications.

Gaschnig (1979, Pg 40) stated that: "clearly the values obtained by this [profile] depend on the number of samples on which the [profile] was based...". (Gaschnig's sample was based on 11,488 out of 60 billion possible combinations, for a sample selection ratio of 1 in six million, but with the same possibility for duplications as our Method A). The important issue is not only choosing a statistically significant number of values to sample (which sample A did), but also sampling randomly over a wide variety from the tctal available (which neither Method A nor Gaschnig did). Method B appears to succeed on both points, providing a wider sample at each level tailored to the shape of the state space, and appearing to capture the average statistical behavior of the heuristic.

The profiles gathered using Method B are referred to as "Source Profiles" since their data was chosen as the basis for the simulation experiments described later. (The Source Profiles for K1, K2, and K3 are presented in the next section.) This name is used to distinguish them from the "Run Profiles" (which are useful for reasons described later) that depict the behavior of a heuristic in application, and tend to be narrower in scope, and therefore not useful as a statistical base for our work.

3. DESCRIPTION OF SOURCE PROFILE GRAPHS

Figures 8.2 through 8.4 present a graphical representation of the source profiles using two distinct forms. The first graph provided is a two-dimensional view of the profile showing KMIN, KMEAN, KMAX, and includes standard deviation information. A line called "optimal" was included for reference, which corresponds to the entity K*, or the value that a perfectly informed heuristic would return. The second pair of graphs shows a three-dimensional view of the same heuristic's profile. The height of the peaks correspond to the frequency each value was encountered, and illustrates the actual distribution of the values that the heuristic calculated. Two views of the same graph are given, one from the front and one from the side of the graph in order to improve the reader's perspective.

Figure 8.2
Source Profile for K1

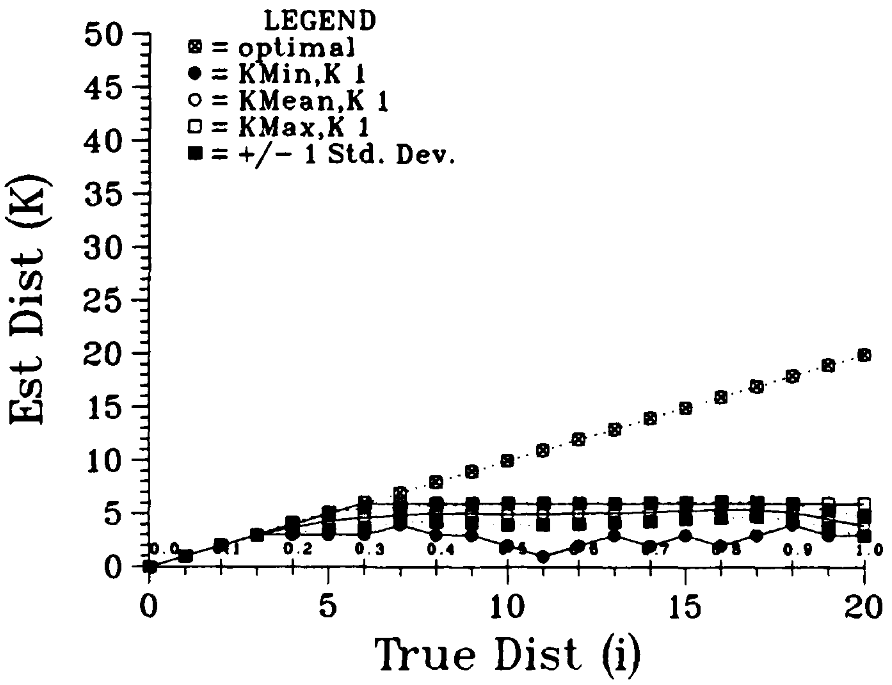


Figure 8.2a Two-Dimensional View

Figure 8.2 (Cont)
Source Profile for K1
Three-Dimensional Views

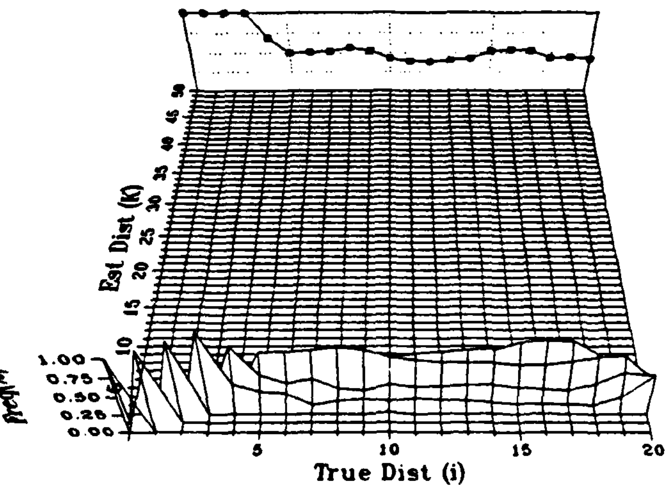


Figure 8.2b Front Perspective

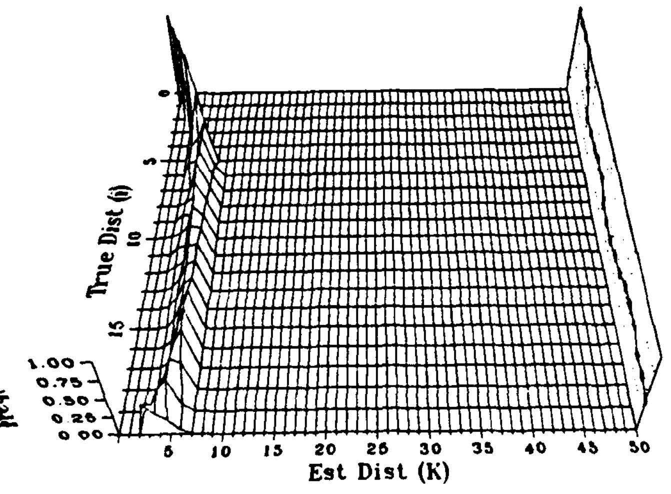


Figure 8.2c Side Perspective

Figure 8.3
Source Profile for K2

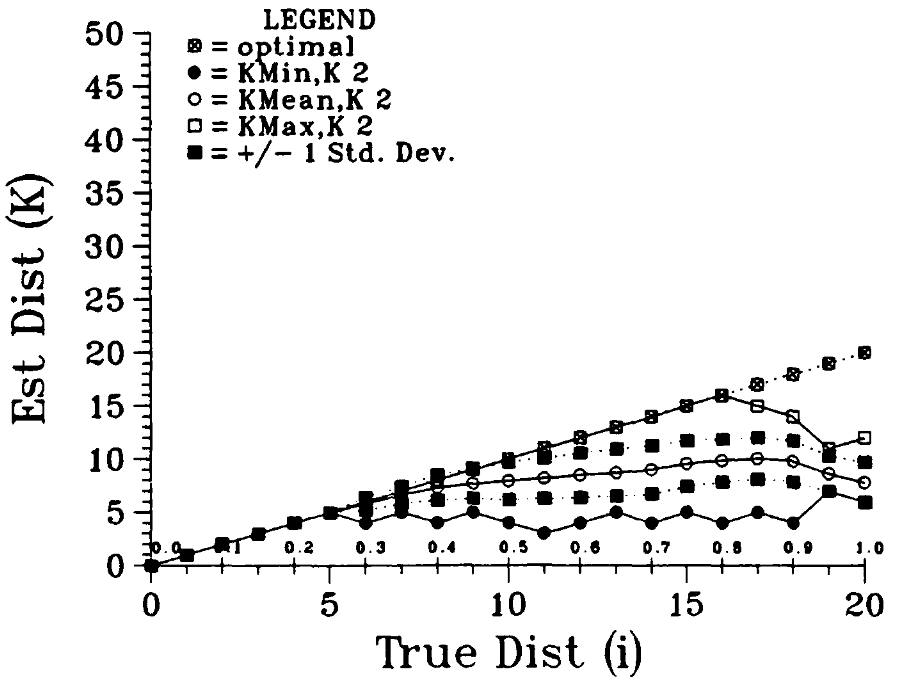


Figure 8.3a Two-Dimensional View

Figure 8.3 (Cont)
Source Profile for K2
Three-Dimensional Views

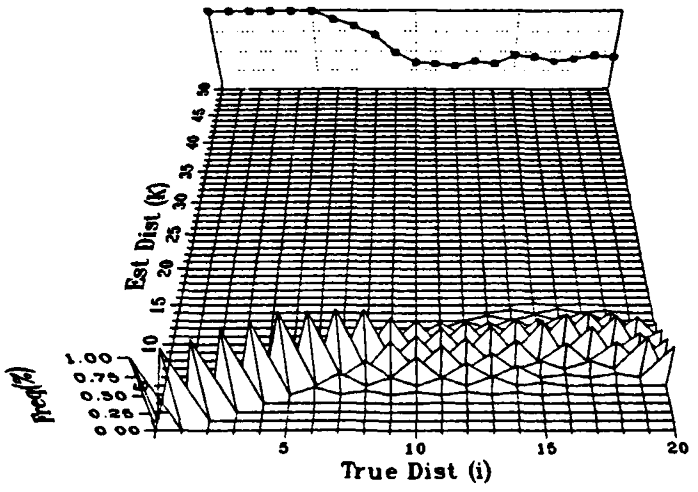


Figure 8.3b Front Perspective

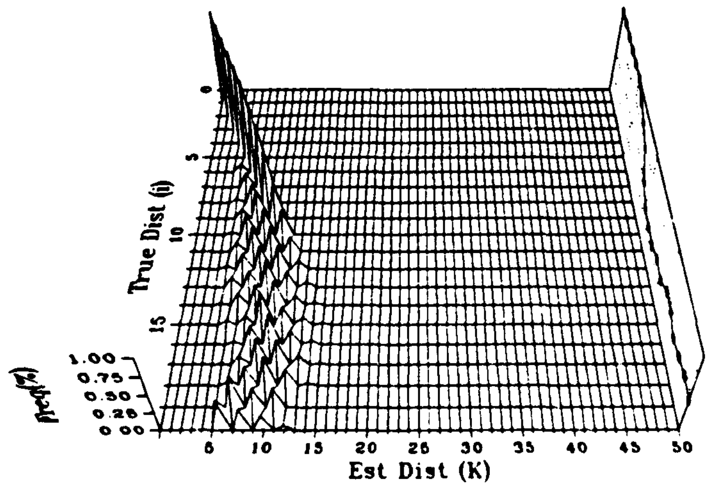


Figure 8.3c   Side Perspective

Figure 8.4
Source Profile for K3

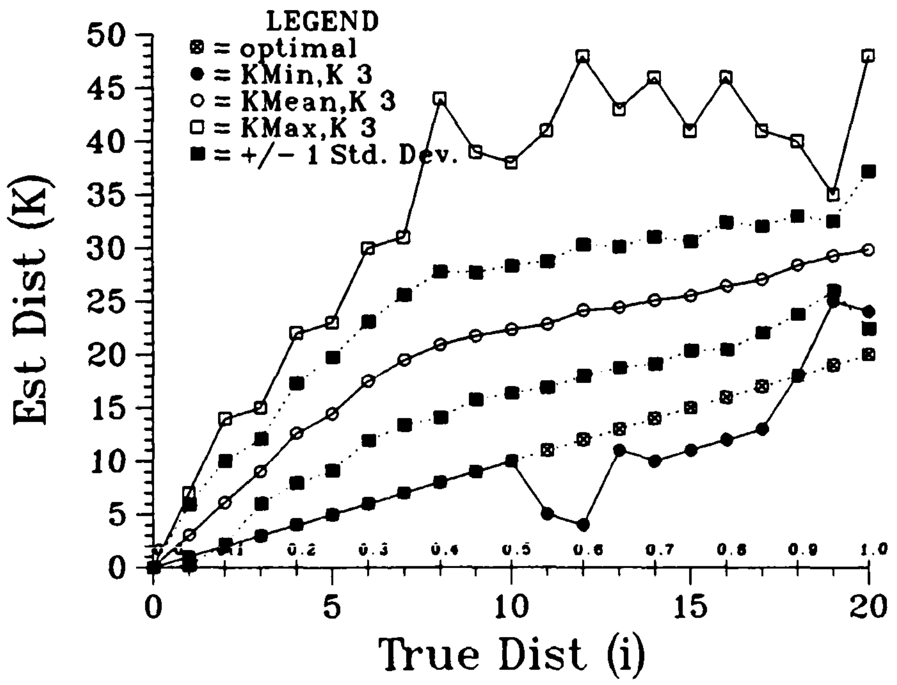


Figure 8.4a Two-Dimensional View

Figure 8.4 (Cont)
Source Profile for K3
Three-Dimensional Views

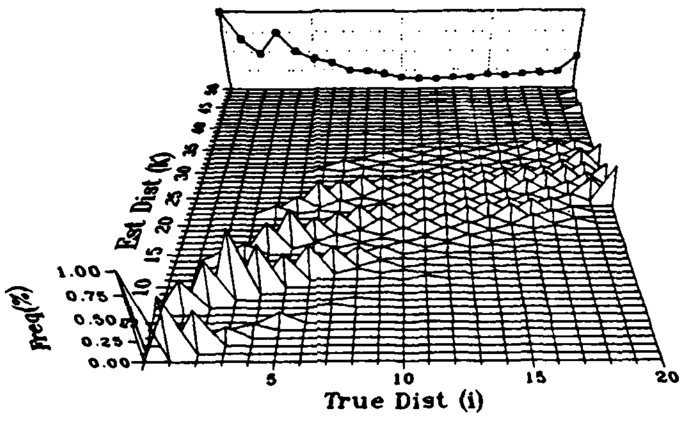


Figure 8.4b Front Perspective

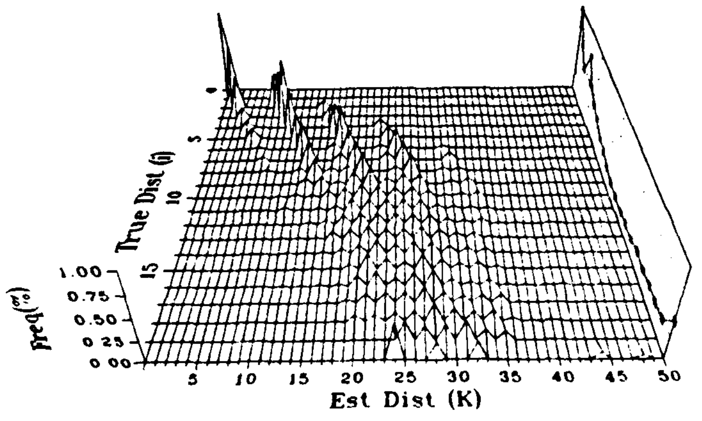


Figure 8.4c Side Perspective

C. CONTRIVED HEURISTICS

Having generated a set of accurate profiles for K1, K2, and K3, we could proceed to see how accurately they captured the essence of the actual heuristic. The basic technique was described earlier, and this section focuses on the contrived heuristics that were implemented. The primary goal of the simulation is to see if equivalent heuristics (as per their profile) result in identical performance in terms of the number of nodes expanded and solution path lengths found.

Essentially, the profile supplies the information gathered on the actual heuristic, and the contrived heuristic decides how to use that information (i.e. what distribution of values to return). While several alternatives were available to us, we wanted to carefully choose the distribtion that the contrived heuristic would assume so that our objectives could be preserved. We wanted contrived heuristics that would behave realistically (with some variation), which led us to choose Options 6 and 7 (described in Section A.4). However, for comparisons sake, we also chose option 8, in order to see the results of using a dramatically different distribution. Each of these contrived heuristics are described below.

1. NORMAL DISTRIBUTION

The contrived heuristics using this distribution return a random number in such a way that the distribution of the aggregate values produced during the entire run forms a "normal" (bell-shaped) curve around KMEAN. This distribution depends on the standard deviation of the values in the Source Profile, and distributes the values so that 68% fall within one standard deviation of KMEAN, 93% fall within two standard deviations, 97.8% fall within three standard deviations, and 99.6% fall within four standard deviations of KMEAN.

Note that the emphasis is to focus on KMEAN rather than KMIN and KMAX, and that some of the values returned may actually be outside the limits KMIN and KMAX in the Source Profile. However, this contrived heuristic should give a good statistical reproduction of the original heuristic's profile, and therefore, we expect that it should also behave identically in terms of number of nodes expanded and solution path length found.

2. ACTUAL DISTRIBUTION

The contrived heuristic based on this distribution randomly selects values according to the frequency distribution of the values calculated by the actual heuristic. This means that if the actual heuristic calculated the value 10 at distance 15 from the goal 23% of the time, then the contrived heuristic should return that same value with the same frequency over the course of the run when the goal is 15 moves away. Values from this distribution remain within the bounds of KMIN and KMAX, and

KMEAN remains virtually the same also.

This contrived heuristic should reproduce the profile from the original heuristic very closely (at least as precisely as is possible), and we expect it to duplicate the performance of the actual heuristic.

3. WORST-CASE DISTRIBUTION

This contrived heuristic provides a different distribution than those described above, and emulates a worst-case behavior model empirically. This distribution deliberately attempts to divert the search process from the ideal path to the goal by over-estimating the distance of the correct nodes, and undercutting (or under-estimating) the values of nodes off the solution path. Since the search algorithm selects nodes with the lowest F-value, this heuristic encourages the search to wander away from the best solution paths. Our implementation of this contrived heuristic returns KMEAN plus one standard deviation if the node being evaluated is on the ideal solution path, and returns KMEAN minus one standard deviation otherwise.

This type of distribution should not deviate greatly from the limits KMIN and KMAX (depending on the size of ST_DEV) but will definitely alter KMEAN. This will provide empirical verification of whether Gaschnig's definition of equivalence (based only on KMIN and KMAX) is sufficient, or whether a full profile is necessary.

4. MECHANICAL ISSUES

Notice that the simulated heuristics need to know the actual distance to the goal in order to look up the appropriate figures in the profile, which would be cheating in the real world because a real heuristic wouldn't know this information. In fact, this is precisely what the heuristic is supposed to be telling us.

Since the simulation requires that the contrived heuristic be provided with the actual distance, a brief discussion of our method of supplying it with this information is in order. There were two methods available to calculate the actual minimal distance between each node during the search and the goal. One method was to invoke the A* algorithm using an admissible F to find the distance, but this was unappealing and was rejected because every node generated would require a separate execution of A*, which was being used already to solve the real problem! Another method entailed generating the state space with the goal as the root using the A* algorithm with an admissible F. Nodes generated during the solution of the start/goal pair correspond to nodes located in this "inverted" state space. The distance to the goal involved searching the inverted state space for that node, and using its G as the value I. (Additional pointers were added to the node structure in order to provide efficient searching of the inverted state space.) While generating the state space is expensive (2520 nodes), this overhead could be virtually eliminated using a carefully chosen sample, where all of the start/goal pairs use a common goal state. In this situation, the cost of building the inverted state space is paid only once for all 198 problems solved, saving thousands of A* executions.

D. EMPIRICAL RESULTS

Each of the contrived heuristics described above was run using the three source profiles (K1, K2, and K3) as its information source (giving nine simulated heuristics in all), using the Weighted A* Graph Search algorithm with G as the discriminator, and at weights 0.2, 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0. In order to distinguish between so many heuristics, we extended the shorthand naming convention used by the actual heuristics (i.e. K1, K2, K3). Table 8.3 provides a legend indicating the name and type for each of the heuristics examined in this empirical study.

TABLE 8.3
Legend of Heuristic Names

Name	Actual or Contrived	Description
K1	Actual	Number of tiles misplaced
K2	Actual	Manhattan Distance
K3	Actual	Enhanced Manhattan Distance
K4	Simulated K1	Normal
K5	Simulated K2	Normal
K6	Simulated K3	Normal
K7	Simulated K1	Histogram
K8	Simulated K2	Histogram
K9	Simulated K3	Histogram
K10	Simulated K1	Worst-case
K11	Simulated K2	Worst-case
K12	Simulated K3	Worst-case

Since there are three contrived heuristics using the profile created from each actual heuristic, it is frequently convenient to refer to this set by group, and the following notation will be adopted: the entire group of four (one actual heuristic and its three contrived couterparts) will be referred to as a 'Set', with Set K1 referring to the heuristics K1, K4, K7, and K10 (since K4, K7, and K10 borrow K1's Source Profile); Set K2 refers to the heuristics K2, K5, K8, and K11; and Set K3 refers to K3, K6, K9, and K12. When an entire set is not desired, subsets will use the following notation: K1/4/7 meaning heuristics K1, K4 and K7, and K2/5/8 meaning heuristics K2, K5, and K8, etc.

The following pages contain graphs representing the results of the simulation experiments. First, we present the run profiles generated by each heuristic during the solution of the problem set, followed by the graphs depicting their performance in terms of number of nodes expanded and length of the solution path found.

1. RUN PROFILES

Figures 8.5 through 8.16 show the performance of the 12 heuristics (Run Profiles) during the solution of the 198 problem pairs, including the observed KMIN, KMEAN, KMAX, and standard deviation values. (Note that a run profile is distinct from the Source Profiles used by the simulation.) As in the graphs for the source profiles, the run profiles are presented using a two-dimensional view coupled with a pair of three-dimensional views to assist in picturing the distribution of the values returned by the heuristic during the solution of the problem set. *The run profile gives* insight into how the contrived heuristic behaved in comparison to the Source Profile from which it was based.

Figures 8.5, 8.6, and 8.7 show the run profiles for the actual heuristics K1, K2, and K3. Figures 8.8, 8.9, and 8.10 show the run profiles for the contrived heuristics K4, K5, and K6, which were based on a normal distribution about the Source Profile of K1, K2, and K3, respectively, and show the 'bell' curve of values rising from the plane in the three dimensional graphs. Comparing Figures 8.8a, 8.9a, and 8.10a to their Source Profile counterparts (Figures 8.2a, 8.3a, and 8.4a, respectively), one can observe that although KMEAN is closely duplicated, KMIN and KMAX are not the same.

The run profiles for contrived heuristics K7, K8, and K9 (Figures 8.11, 8.12, and 8.13) are interesting because they duplicated their Source Profiles (see Figures 8.2 through 8.4), indicating that they did indeed mimick the aggregate statistical behavior of the actual heuristic.

The run profiles for K10, K11, and K12 (Figures 8.14, 8.15, and 8.16) show the peaks surrounding KMEAN, and show by their height that most of the values returned were for the "off-the-path" nodes and that very few "on-path" nodes were expanded. This is confirmed by observing the two-dimensional versions, where KMEAN is almost superimposed on top of KMIN, yet KMAX hovers high above.

Figure 8.5
Run Profile for K1

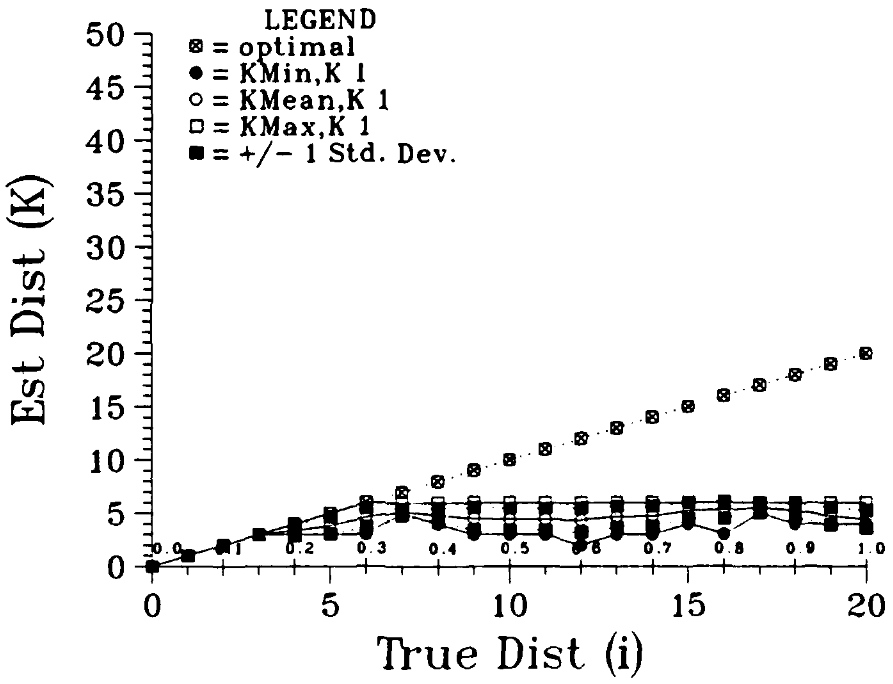


Figure 8.5a Two-Dimensional View

Figure 8.5 (Cont)
Run Profile for K1
Three-Dimensional Views

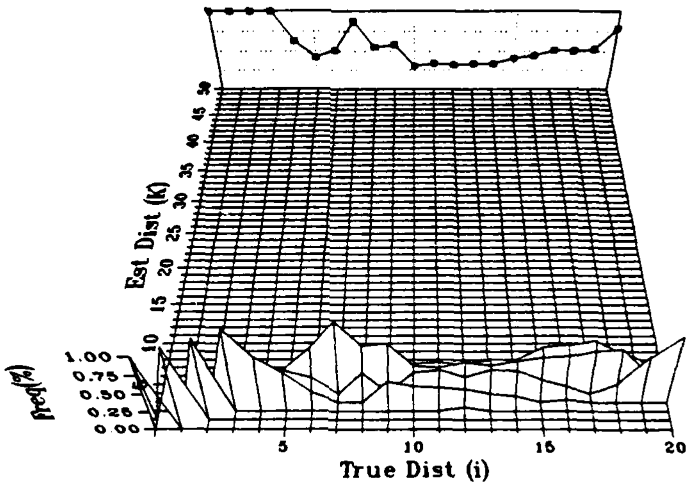


Figure 8.5b Front Perspective

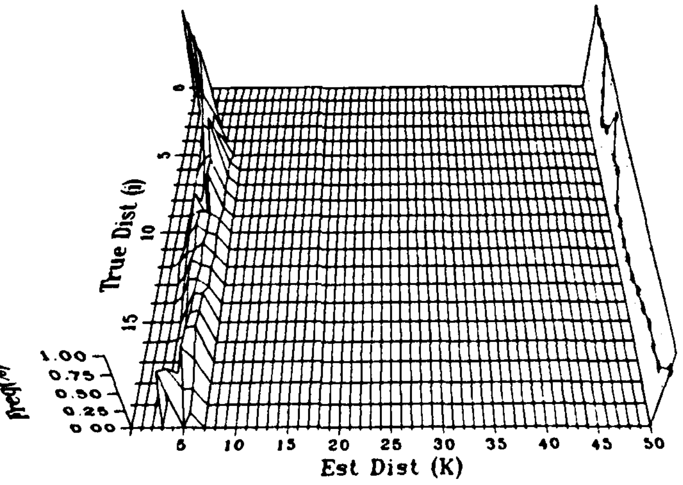


Figure 8.5c Side Perspective

Figure 8.6
Run Profile for K2

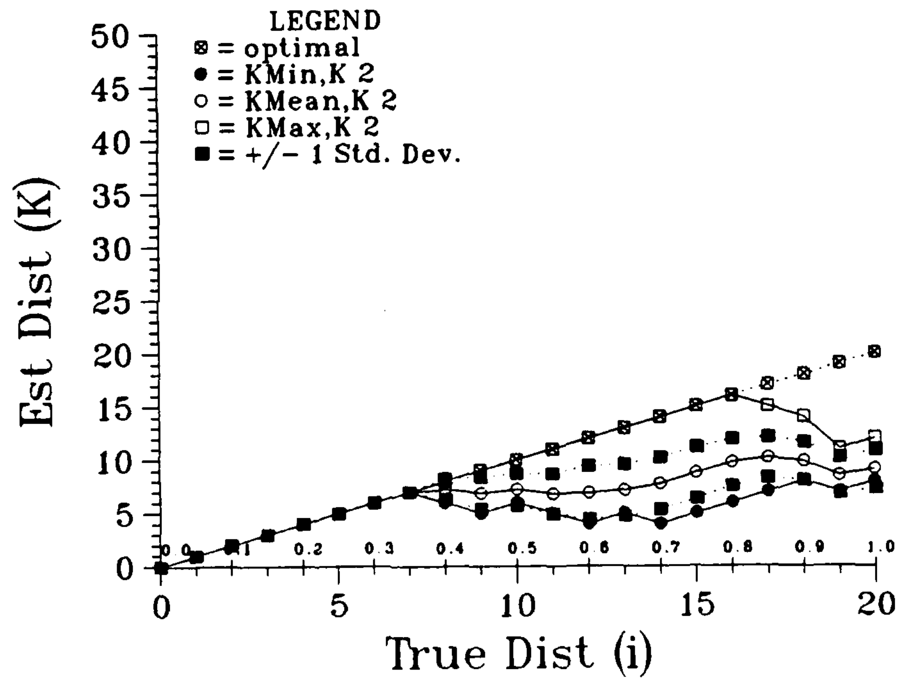


Figure 8.6a Two-Dimensional View

Figure 8.6 (Cont)
Run Profile for K2
Three-Dimensional Views

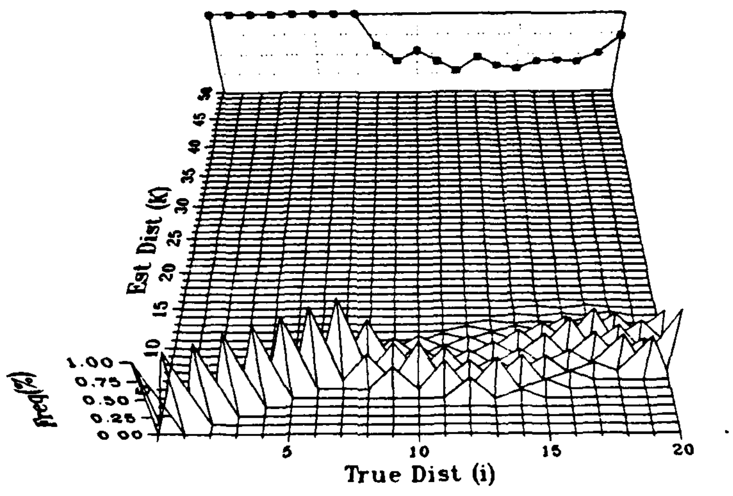


Figure 8.6b Front Perspective

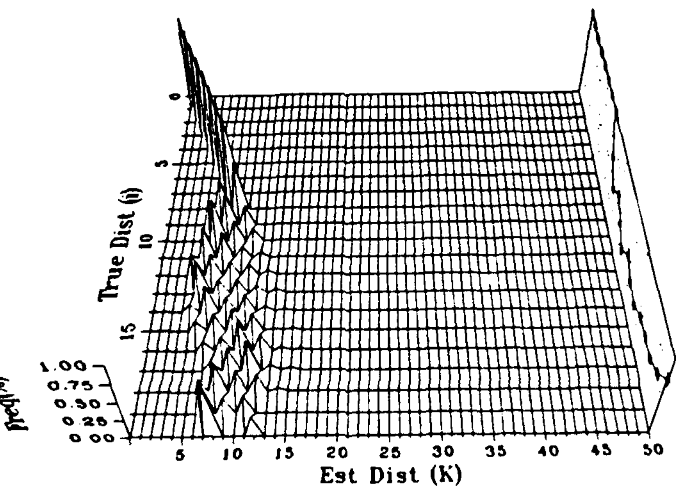


Figure 8.6c Side Perspective

Figure 8.7
Run Profile for K3

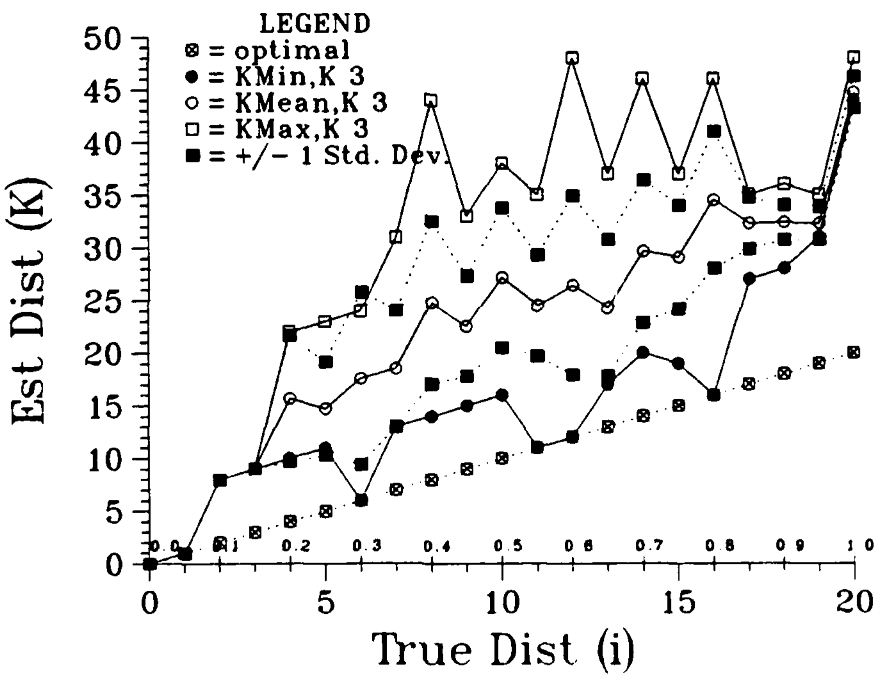


Figure 8.7a   Two-Dimensional View

Figure 8.7 (Cont)
Run Profile for K3
Three-Dimensional Views

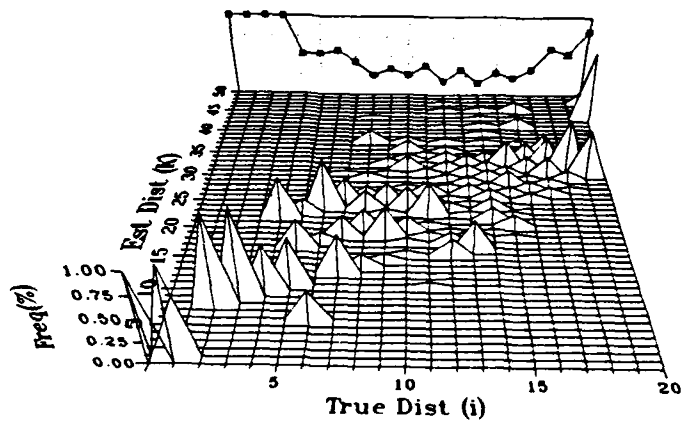


Figure 8.7b Front Perspective

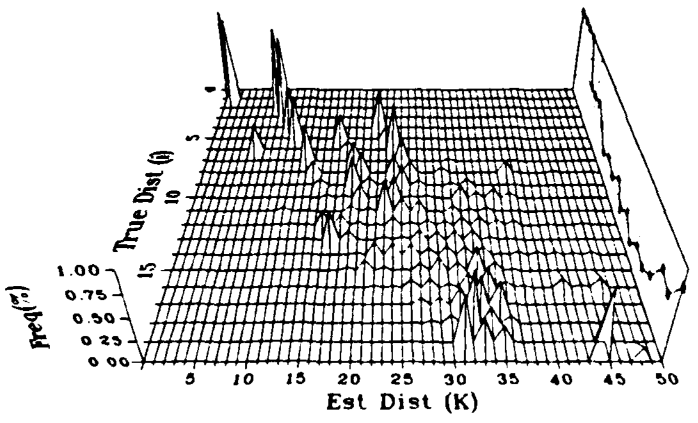


Figure 8.7c Side Perspective

Figure 8.8
Run Profile for K4

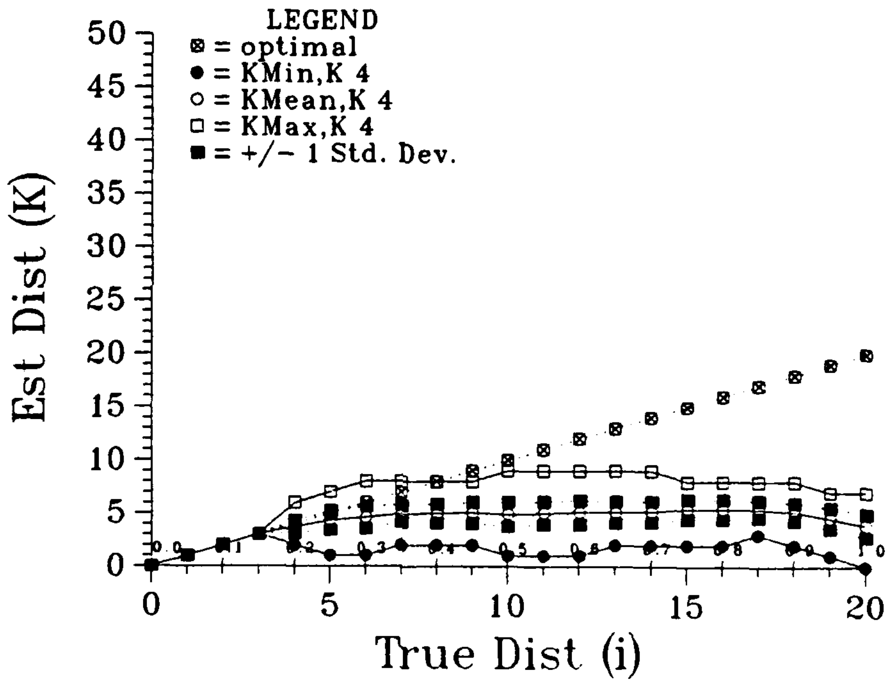


Figure 8.8a Two-Dimensional View

Figure 8.8 (Cont)
Run Profile for K4
Three-Dimensional Views

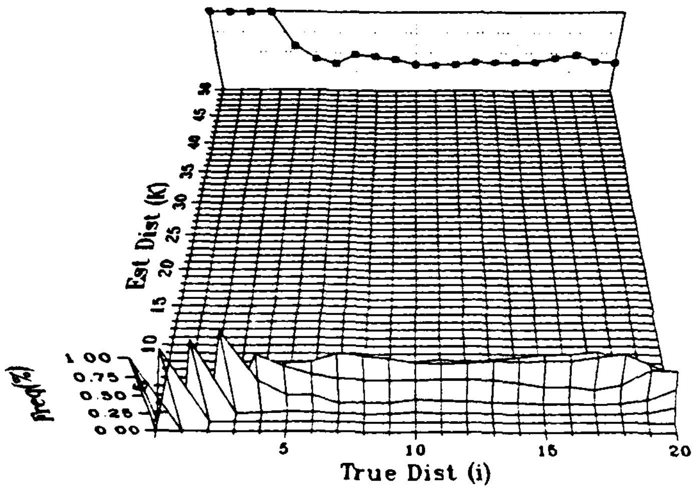


Figure 8.8b Front Perspective

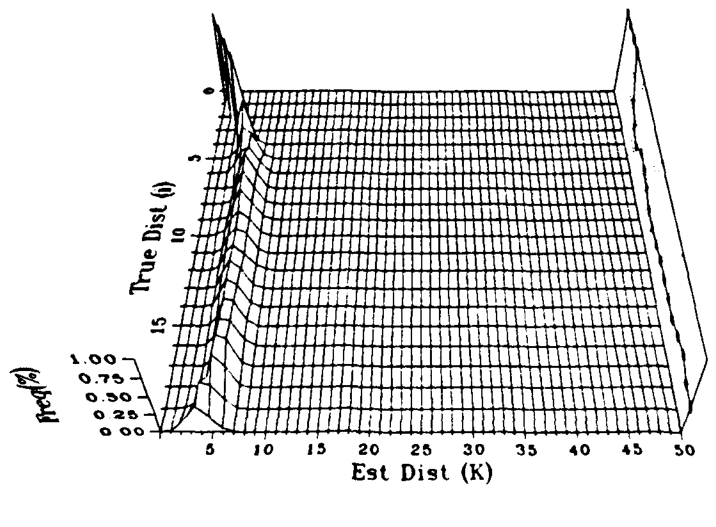


Figure 8.8c Side Perspective

Figure 8.9
Run Profile for K5

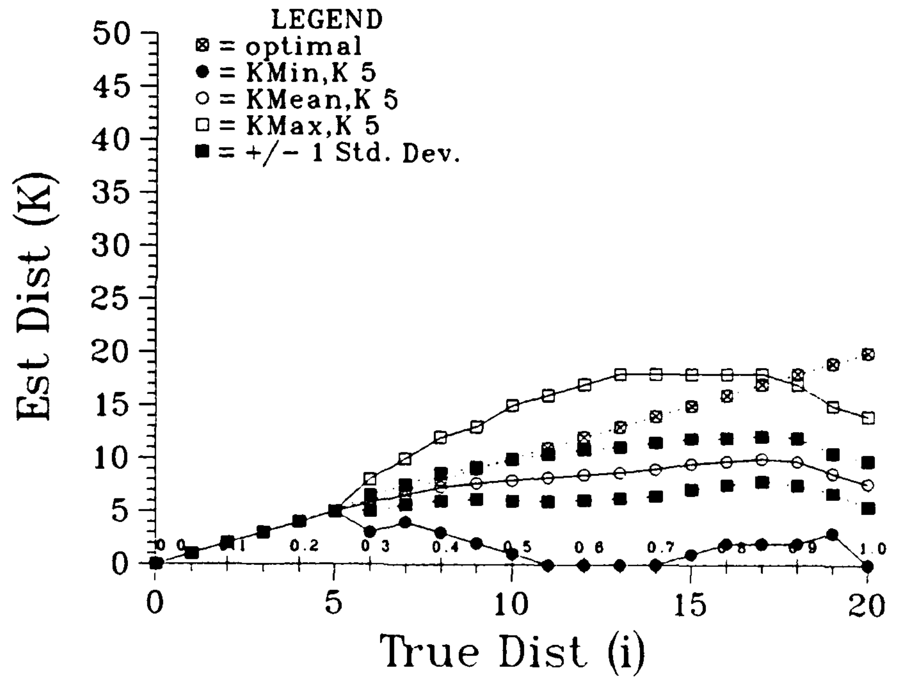


Figure 8.9a Two-Dimensional View

Figure 8.9 (Cont)
Run Profile for K5
Three-Dimensional Views

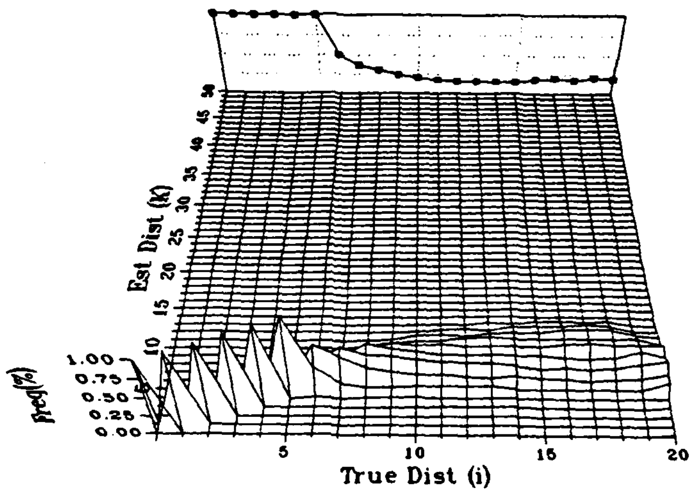


Figure 8.9b Front Perspective

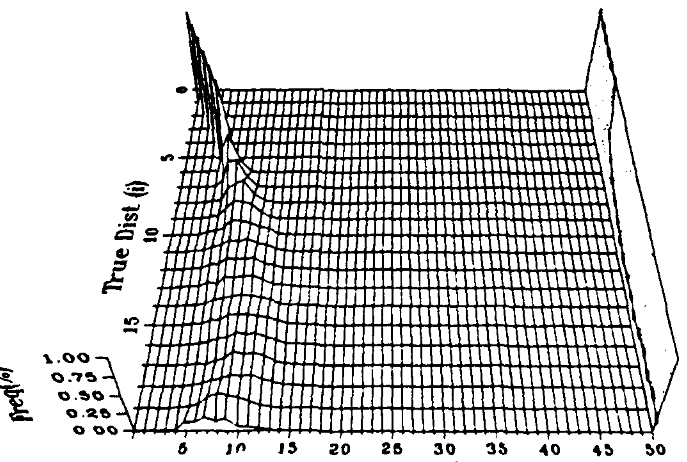


Figure 8.9c Side Perspective

Figure 8.10
Run Profile for K6

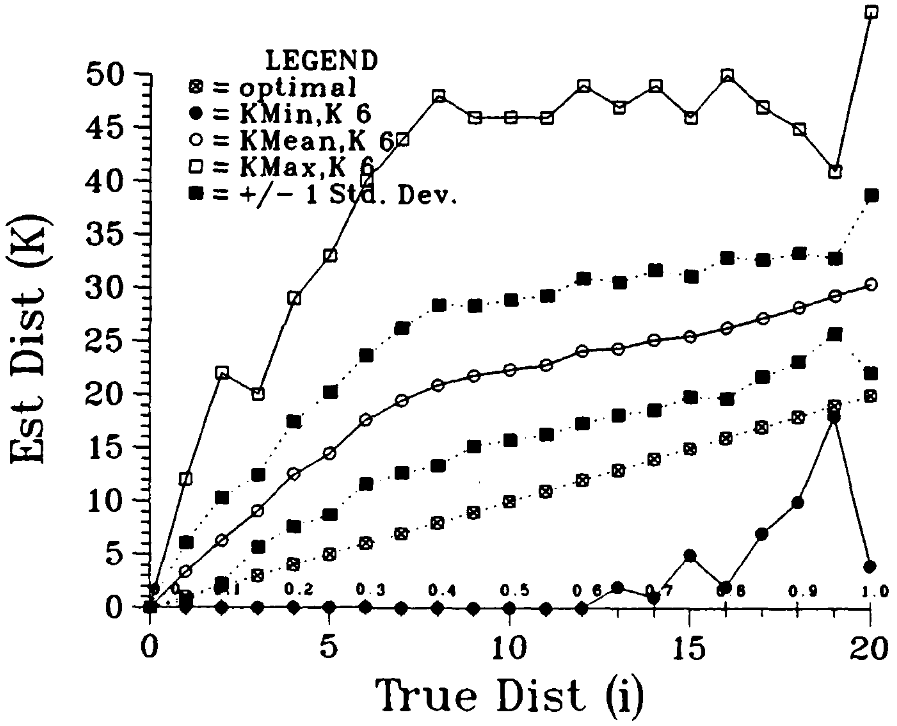


Figure 8.10a Two-Dimensional View

Figure 8.10 (Cont)
Run Profile for K6
Three-Dimensional Views

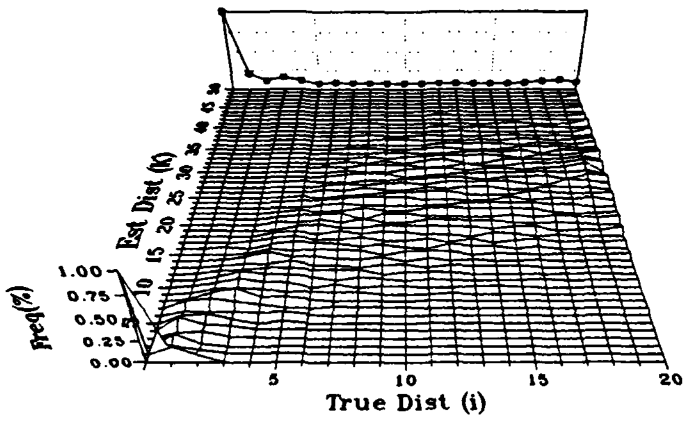


Figure 8.10b Front Perspective

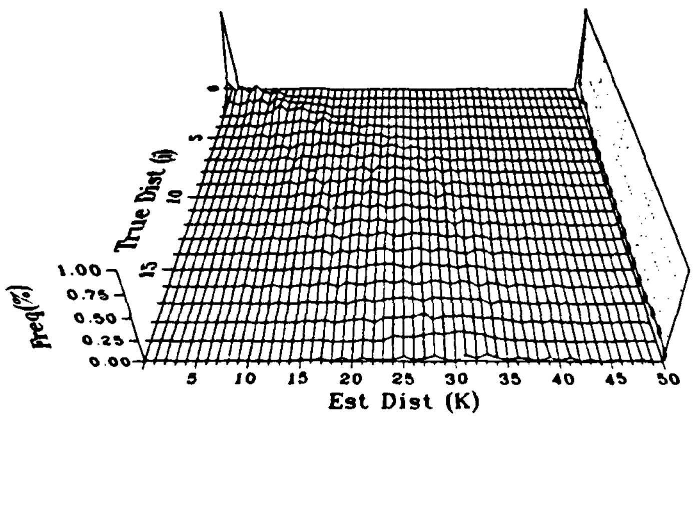


Figure 8.10c Side Perspective

Figure 8.11
Run Profile for K7

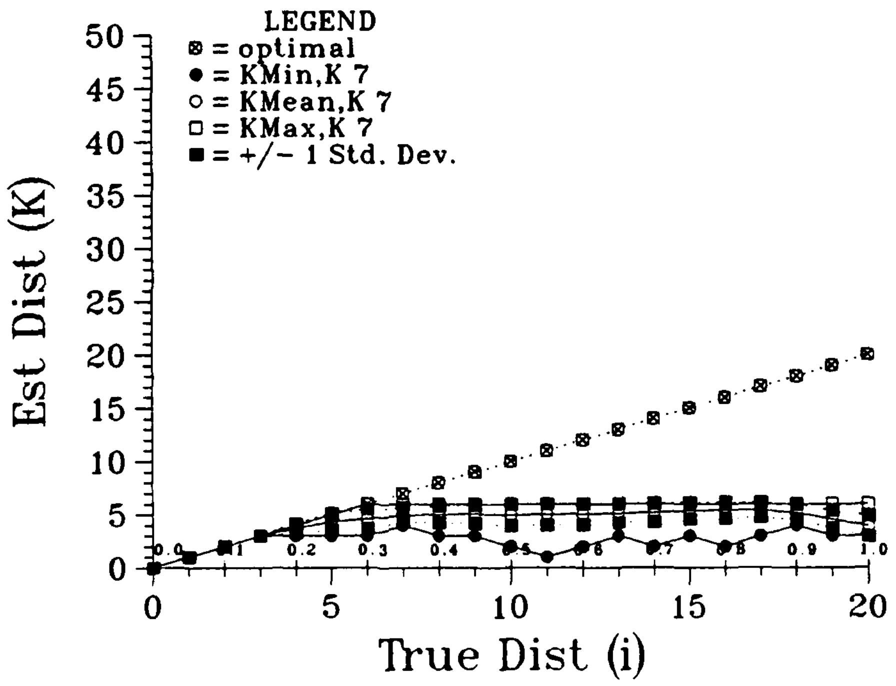


Figure 8.11a Two-Dimensional View

Figure 8.11 (Cont)
Run Profile for K7
Three-Dimensional Views

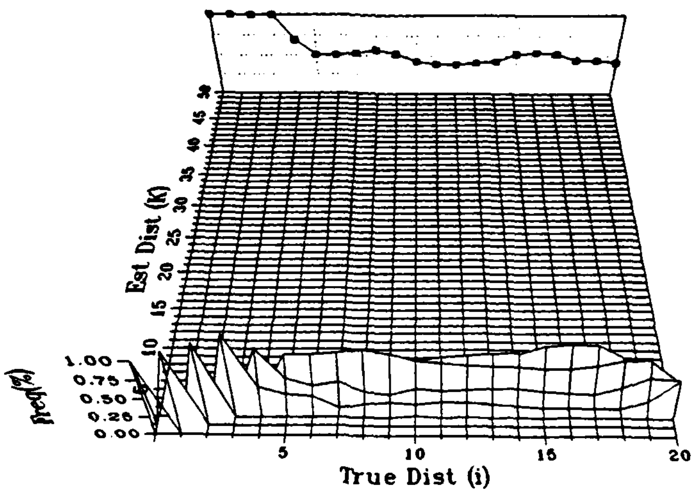


Figure 8.11b Front Perspective

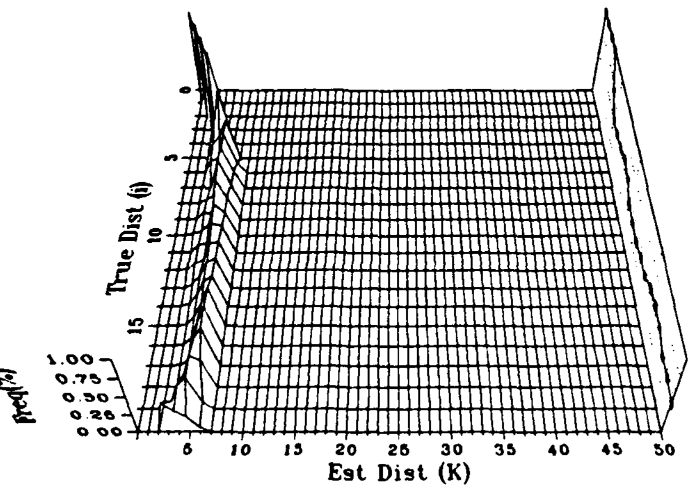


Figure 8.11c Side Perspective

Figure 8.12
Run Profile for K8

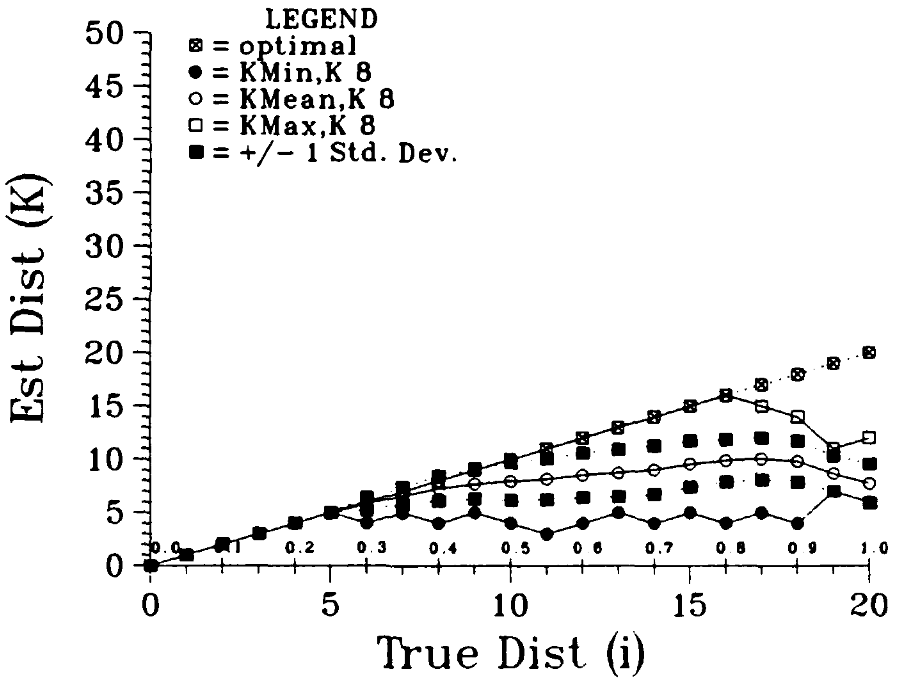


Figure 8.12a Two-Dimensional View

Figure 8.12 (Cont)
Run Profile for K8
Three-Dimensional Views

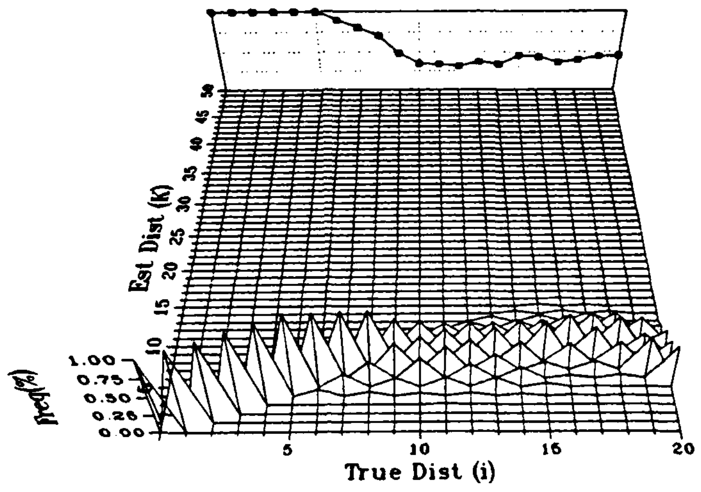


Figure 8.12b Front Perspective

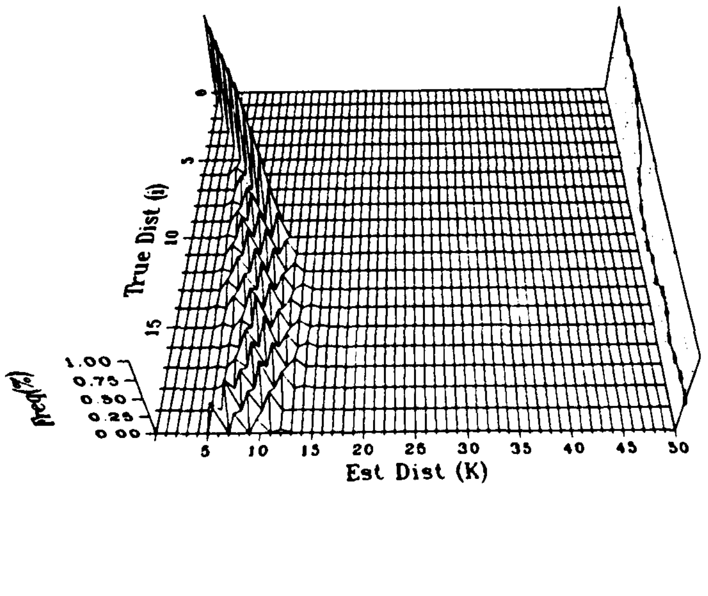


Figure 8.12c Side Perspective

Figure 8.13
Run Profile for K9

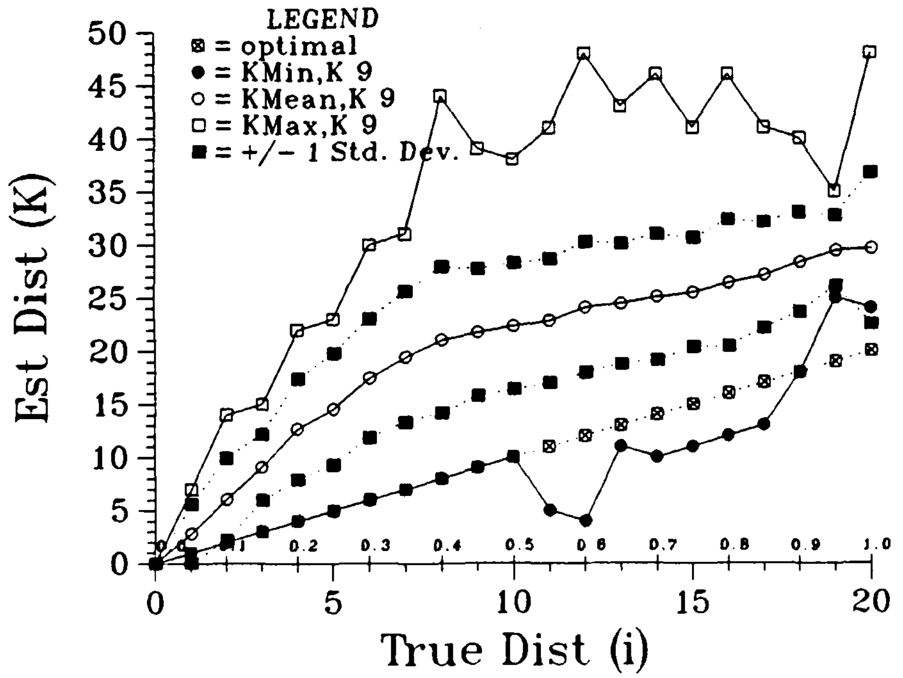


Figure 8.13a Two-Dimensional View

Figure 8.13 (Cont)
Run Profile for K9
Three-Dimensional Views

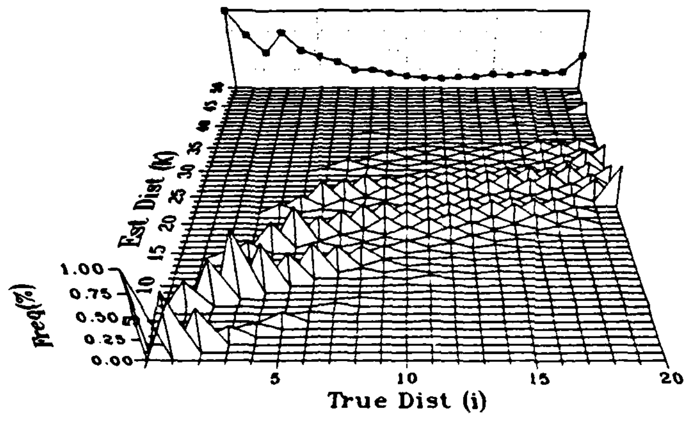


Figure 8.13b Front Perspective

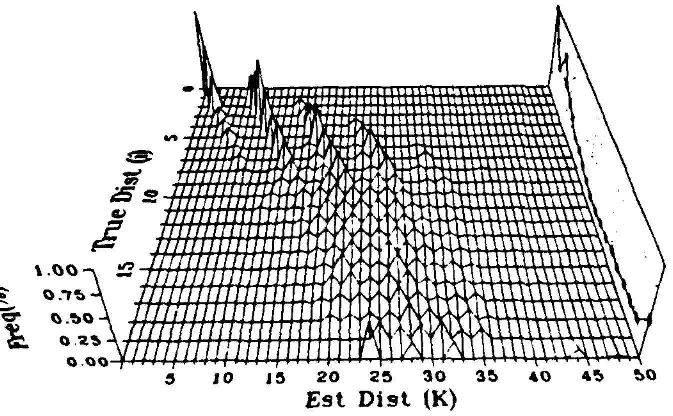


Figure 8.13c Side Perspective

Figure 8.14
Run Profile for K10

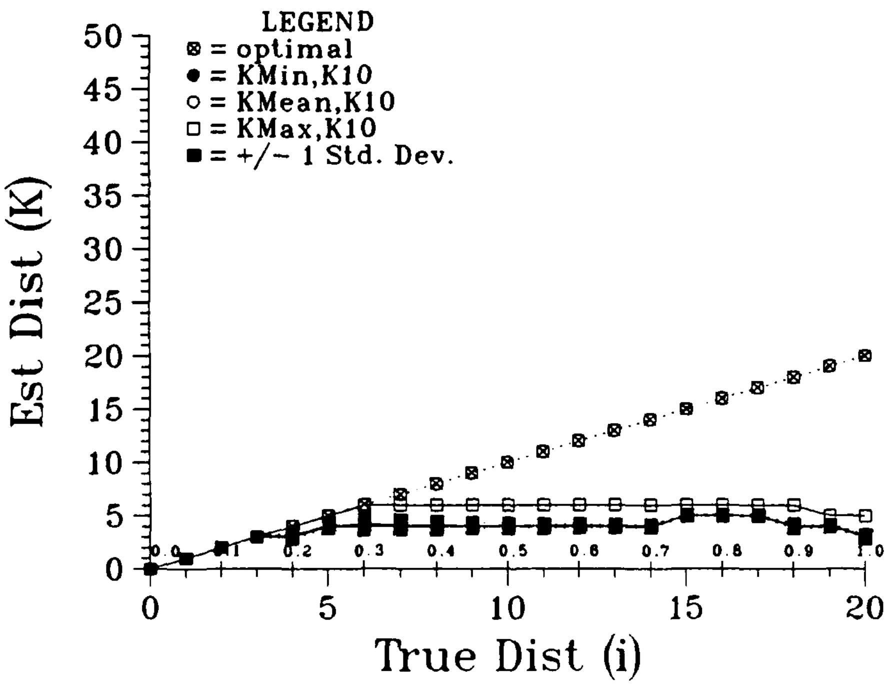


Figure 8.14a Two-Dimensional View

Figure 8.14 (Cont)
Run Profile for K10
Three-Dimensional Views

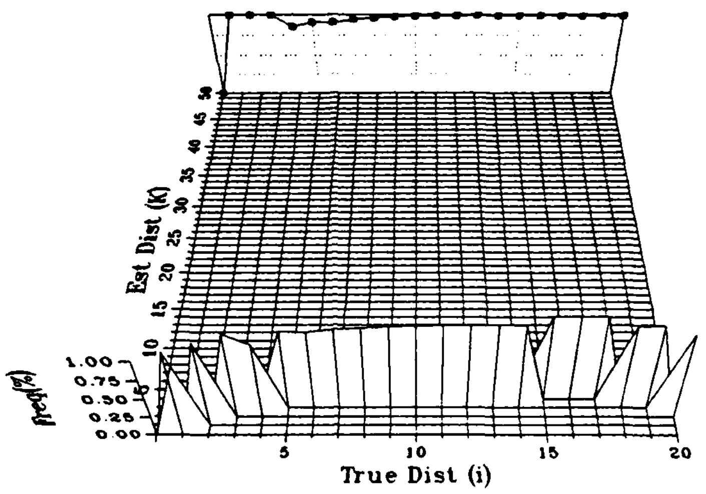


Figure 8.14b Front Perspective

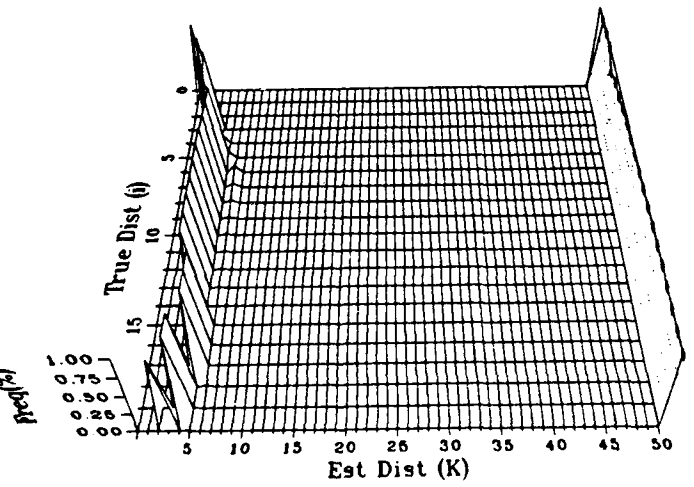


Figure 8.14c Side Perspective

Figure 8.15
Run Profile for K11

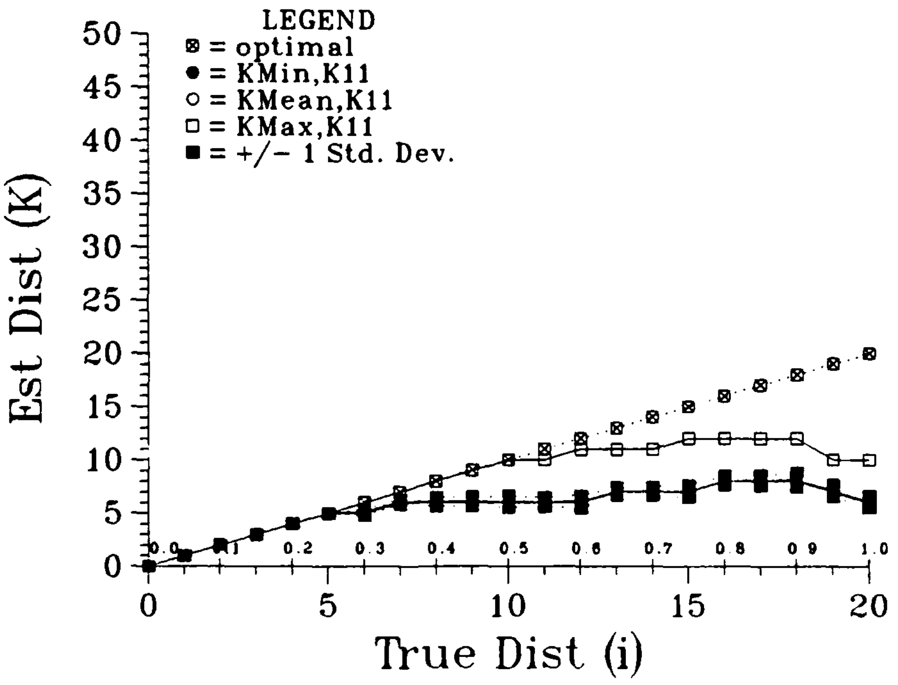


Figure 8.15a   Two-Dimensional View

Figure 8.15 (Cont)
Run Profile for K11
Three-Dimensional Views

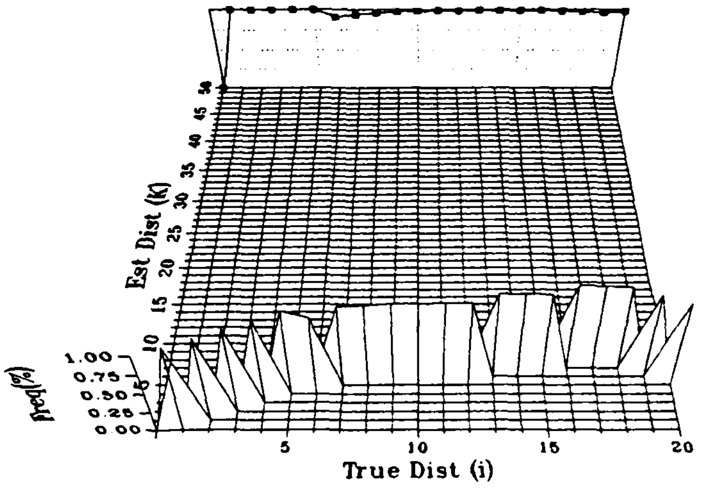


Figure 8.15b Front Perspective

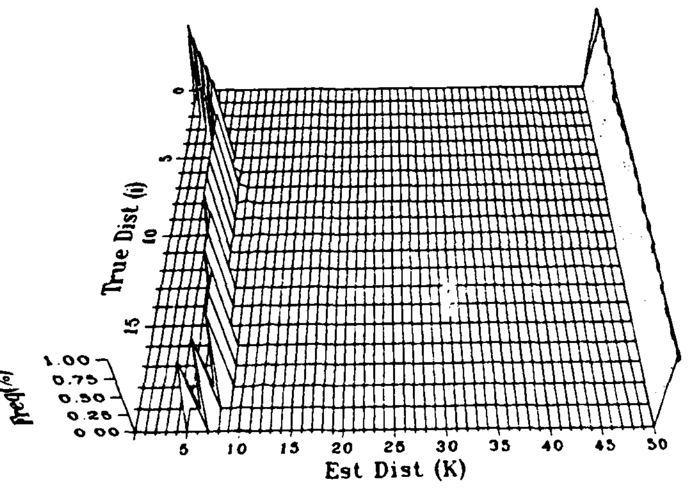


Figure 8.15c Side Perspective

Figure 8.16
Run Profile for K12

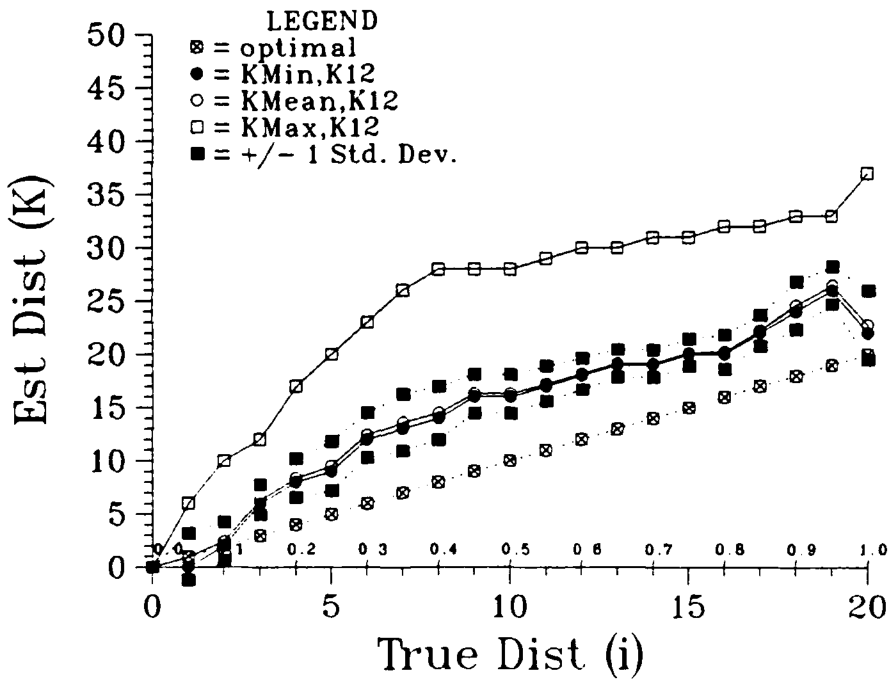


Figure 8.16a Two-Dimensional View

Figure 8.16 (Cont)
Run Profile for K12
Three-Dimensional Views

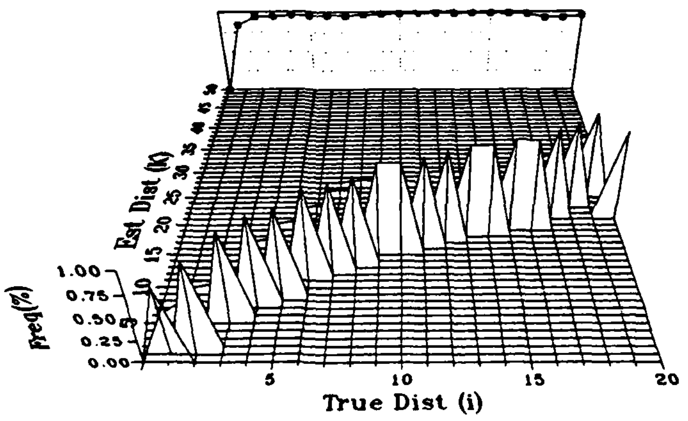


Figure 8.16b Front Perspective

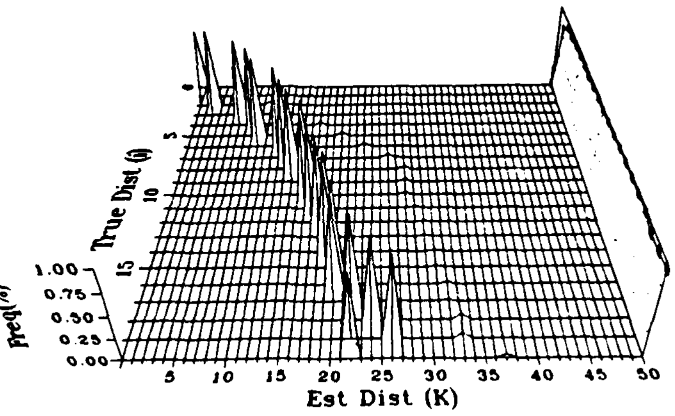


Figure 8.16c Side Perspective

2. COMPLEXITY PERFORMANCE RESULTS

Figures 8.17 through 8.22 graphically present the performance results of the simulation experiments in terms of nodes expanded versus the depth of the goal (Figures 8.17 to 8.19) and solution path length versus depth of goal (Figures 8.20 to 8.22). In order to save space and enhance comparison between the contrived heuristics and their actual counterparts, each graph contains four performance curves: one for each of the three contrived heuristics, and a fourth curve for the actual heuristic from whose profile the contrived heuristics were based.

Each plot presents the performance of the set of four heuristics at one weight. The results are rich with information, so only the most significant phenomenon will be highlighted. First we focus on the trends observed from the Normal and Actual distributions because their behavior was similar enough to one another to merit being considered together. We will then examine Worst-Case performance separately.

a. NORMAL AND ACTUAL DISTRIBUTION RESULTS

The performances of Normal and Actual contrived heuristics were quite similar to each other. In the graphs that follow, their curves tend to follow each other around rather independently of the other curves on the graph. For Set K1 and Set K2 at the lower weights (Figures 8.17a, 8.17b, and 8.18a), and for Set K3 at all weights (Figures

8.19a-f) they also coincide very closely with the performance of the actual heuristic. However, when one deviates, the other is found nearby, as shown in Figures 8.18c-f and 8.17e-f.

1. RANGE OF EFFECTIVENESS

The results show that even when the simulation was not very good, there were always ranges of N in which the simulation was very effective. For K1/4/7, this range appears to be for levels 1-5 (Figures 8.17a-f); for K2/5/8, the range appears to be from 1-7 (Figures 8.18a-f); and for K3/6/9, the simulation is not exact at any single N but fairly close over all N (Figures 8.19a-f). Comparing these ranges against the source profiles (Figures 8.2-8.4) exposes some interesting coincidences: Source Profile K1 (Figure 8.2) shows that KMEAN levels off at 5, and Source Profile K2 (Figure 8.3) shows KMEAN leveing off at around 8, but in Source Profile K3 (Figure 8.4), KMEAN never really levels off at all.

This suggests that the actual heuristics have a range within which they are very effective in predicting the distance remaining to the goal, and beyond that range, their estimate is no longer 'intelligent' but merely a bounded guess. In a sense, they display a 'nearsightedness'. K1 is the worst, never exceeding the value 6, and whose mean levels off at 5. When the goal is beyond that range, K1 is incapable of providing a meaningful estimate and instead, gives a guess that corresponds to its range or the limit of its sight. K2 has somewhat better vision, but it too levels off at around 8 or 9, meaning for many of the more distant goals, its average response is still shortsighted and fairly meaningless.

K3 displays a different behavior altogether. Although KMEAN always overestimates K*, it does so in a monotonically increasing fashion. When the goal is 20 moves away, K3 not only has the capacity to provide a meaningful response, where K1 and K1 cannot, it also consistently lowers its estimate as it gets closer to the goal.

The simulation within the range of each heuristic is good at all weights. Beyond this range, since the actual heuristic doesn't return meaningful values, attempting to simulate this proves to be ineffective.

2. TIMING

Figures 8.11 through 8.13 show the run profiles based on the contrived heuristics using the actual frequency distribution (K7, K8, and K9). These are exact duplicates of the Source Profiles (compare to Figures 8.2 to 8.4). This means that the values generated by the contrived heuristics during the solution of the 198 problems had aggregate performances that were exactly like the actual heuristic's profile, and yet the simulation was not so exact (Figures 8.17f, 8.18c-f). What caused these performance variations when the profiles were so accurately preserved? Timing.

While the profile permits precise duplication of the distribution of the actual heuristic for the entire run, it cannot guarantee that the simulated heuristic will respond with the same value that the actual heuristic would have at any given node. The contrived heuristic may have terrible timing and give high values when the actual heuristic would have given low values, or it may have impeccable timing, returning different values but doing so in such a combination that the search process is led directly to the goal. This could be why K2/5/8 at the higher weights perform so much better than the actual heuristic (Figures 8.18d-f). While timing doesn't affect K3/6/9 as dramatically, this may well be the cause of the occasional minor deviations observed in their graphs (Figures 8.19a-f).

3. WEIGHT

Weight also has an impact on the effectiveness of the simulation, where generally we observe very similar results at the lower weights, but becoming less effective as the weight increases. This probably isn't too significant because at low weights the search is essentially breadth-oriented and the H component has only a minor impact on the direction of the search pattern.

Specifically, simulations for K1/4/7 imitated the actual heuristic's performance very closely over all N when weight was less than 0.8 (Figures 8.17a-d), but when the weight exceeded 0.8, the results deviated significantly beyond the range of the heuristic (Figures 8.17e and f). For K2/5/8, the simulation breaks up beyond the heuristic's range when weights greater than 0.6 are used (Figures 8.17c-f). Simulation effectiveness for K3/6/9 (Figures 8.18a-f) seems unaffected by weight.

b. WORST-CASE DISTRIBUTION RESULTS

The Worst-Case simulation produced mixed results. We anticipated seeing more nodes expanded than the actual heuristic produced, since the characteristic of this contrived heuristic is to throw the search pattern away from the true path. For Set K1, we observed that K10 provided an upper bound on the other performance curves in terms of nodes expanded at all weights less than 0.8 (See Figures 8.17a-e). However, at weight 0.9 (Figure 8.17f), the growth at levels (N) 5-6 was super-exponential, jumping from about 7 nodes expanded to over 100, but leveling completely for N>6.

K11 provides similar behavior for Set K2, giving an upper bound for nodes expanded for weights less than 0.8 (Figures 8.18a-d), and, like K10, takes a sudden steep rise at weight 0.8 and 0.9 (Figures 8.18e-f), leveling off after that. An interesting phenomenon is then observed: K11 becomes a lower bound for nodes expanded for Set K2 at levels 12 to 19!

K12 shows the same trend, giving an upper bound at weight 0.2 (Figure 8.19a), and for weights 0.5 through 0.9 (Figures 8.19b-f) taking the sudden, steep rise that subsequently levels off, and like K11, becomes a lower bound for the deeper levels of N. In fact, it becomes near-optimal.

The upper bound results were expected, but the leveling off and lower bound results are curious and deserve some explanation. The steep rise indicates that the search is nearly Breadth-First in nature. The distribution adds or subtracts one standard deviation from KMEAN, which at the lower values of i, represents a sizeable portion of the F-value. As i increases, the impact of one standard deviation added to or taken away from KMEAN becomes less significant, and the search becomes increasingly effective.

Figures 8.20 through 8.22 show the path lengths, and indicate the solution path lengths for K10, K11, and K12 at the middle values of N were very large, tapering off to almost ideal as N increased. This suggests that where the search expanded the full tree, the ideal path was avoided (because it was forced to by the nature of the contrived heuristic), and longer, alternative detours were taken to arrive at the goal.

Figure 8.17a
Simulation Results
Heuristics K1, K4, K7, K10
Weight = 0.2
XMEAN

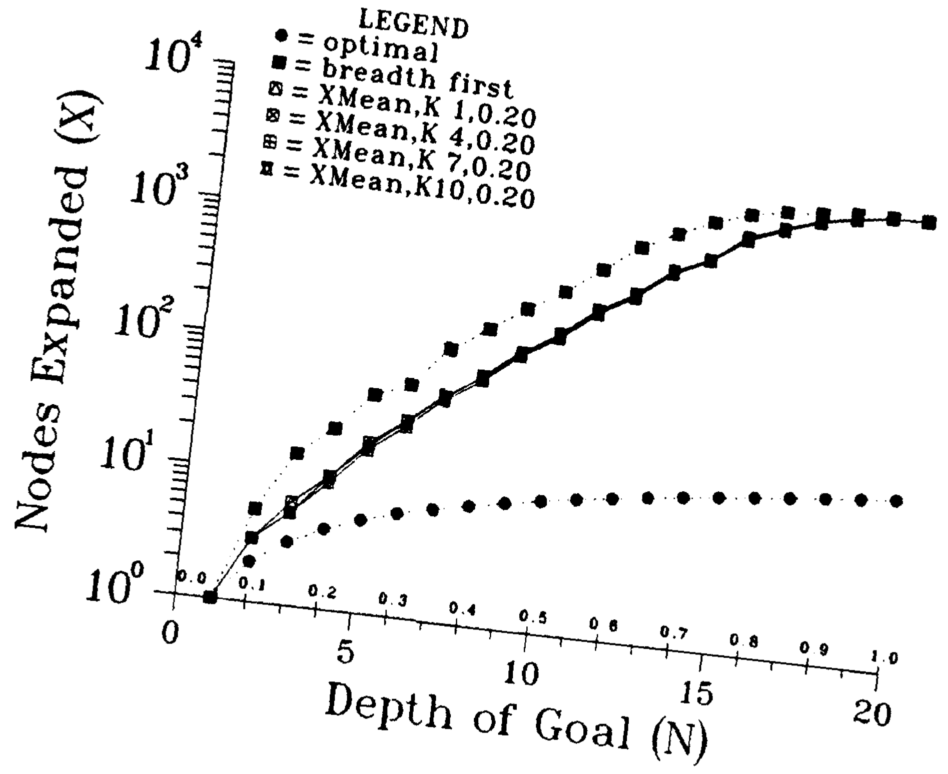


Figure 8.17b
Simulation Results
Heuristics K1, K4, K7, K10
Weight = 0.5
XMEAN

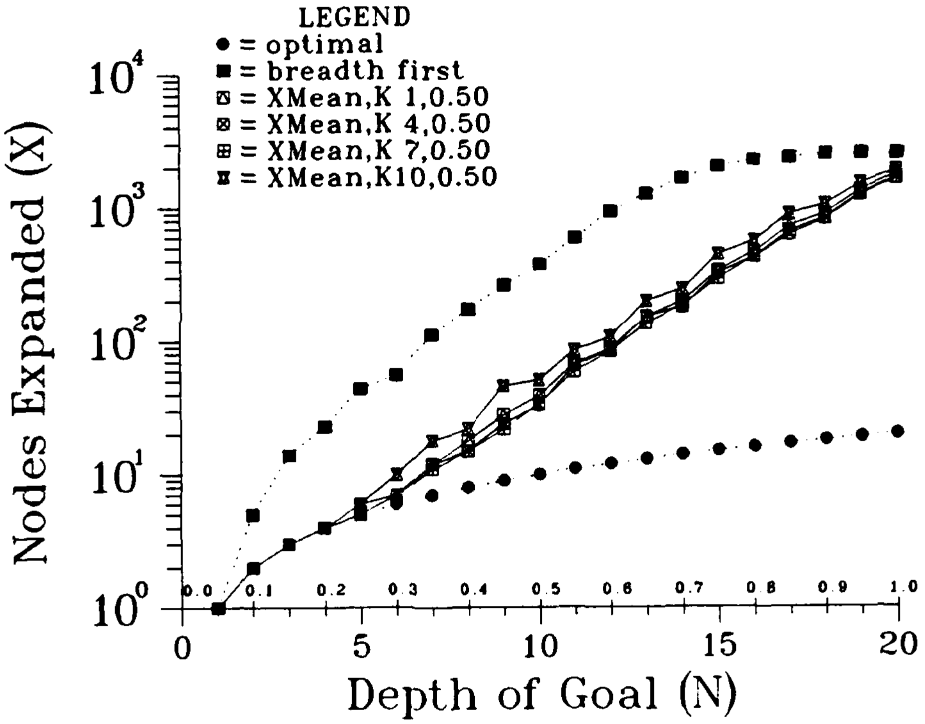


Figure 8.17c
Simulation Results
Heuristics K1, K4, K7, K10
Weight = 0.6
XMEAN

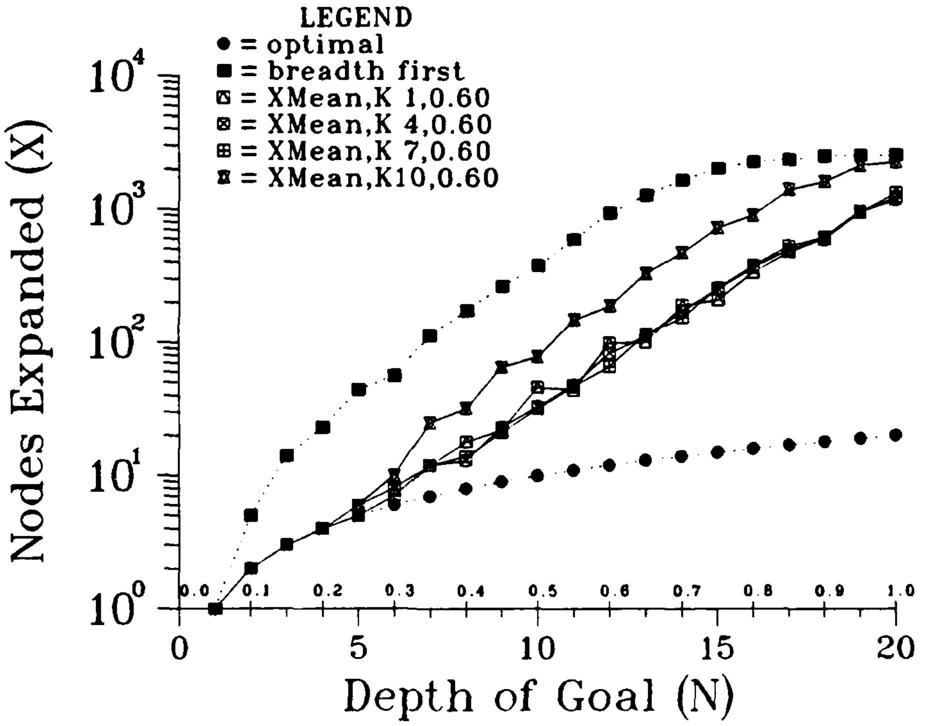


Figure 8.17d
Simulation Results
Heuristics K1, K4, K7, K10
Weight = 0.7
XMEAN

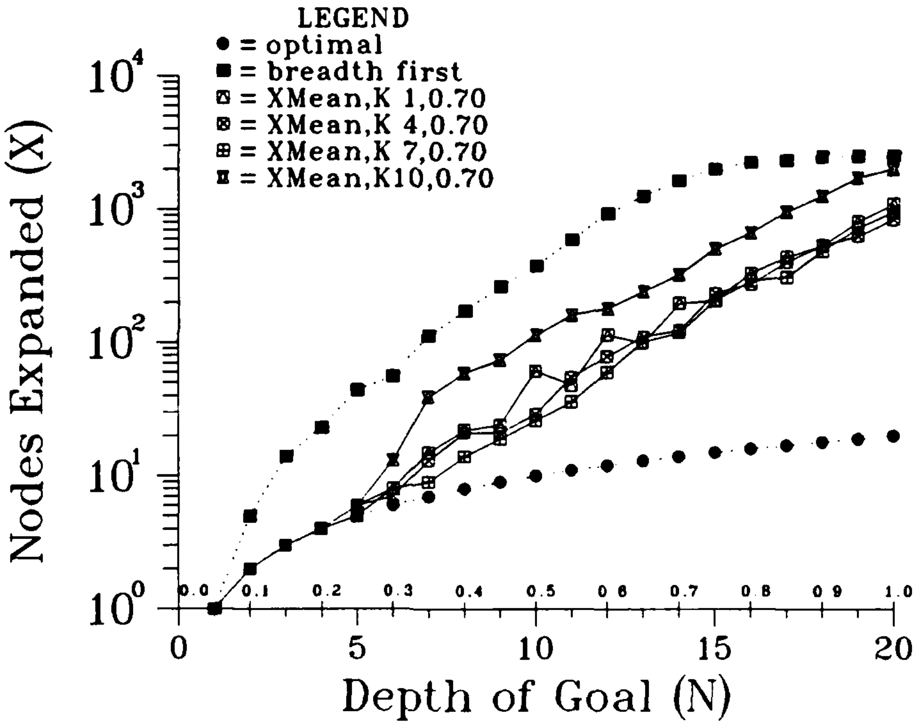


Figure 8.17e
Simulation Results
Heuristics K1, K4, K7, K10
Weight = 0.8
XMEAN

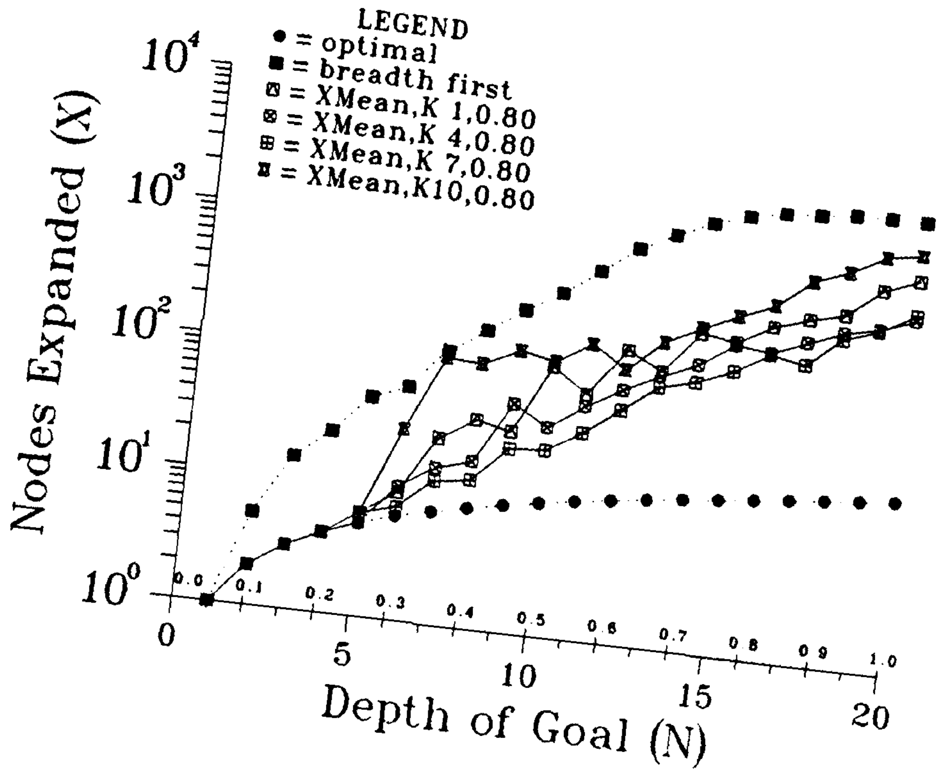


Figure 8.17f
Simulation Results
Heuristics K1, K4, K7, K10
Weight = 0.9
XMEAN

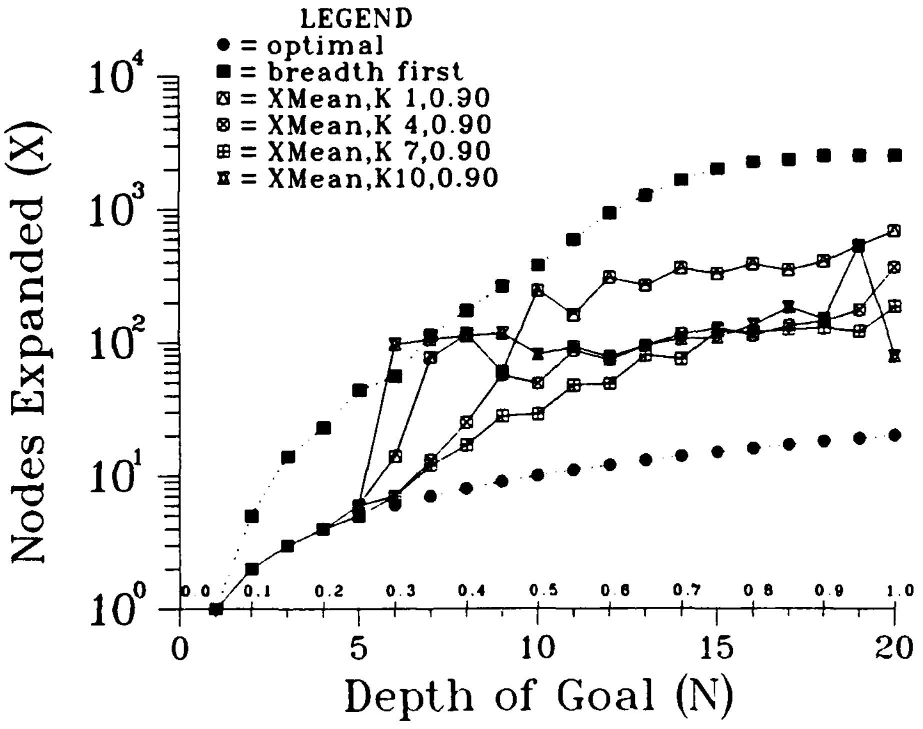


Figure 8.18a
Simulation Results
Heuristics K2, K5, K8, K11
Weight = 0.2
XMEAN

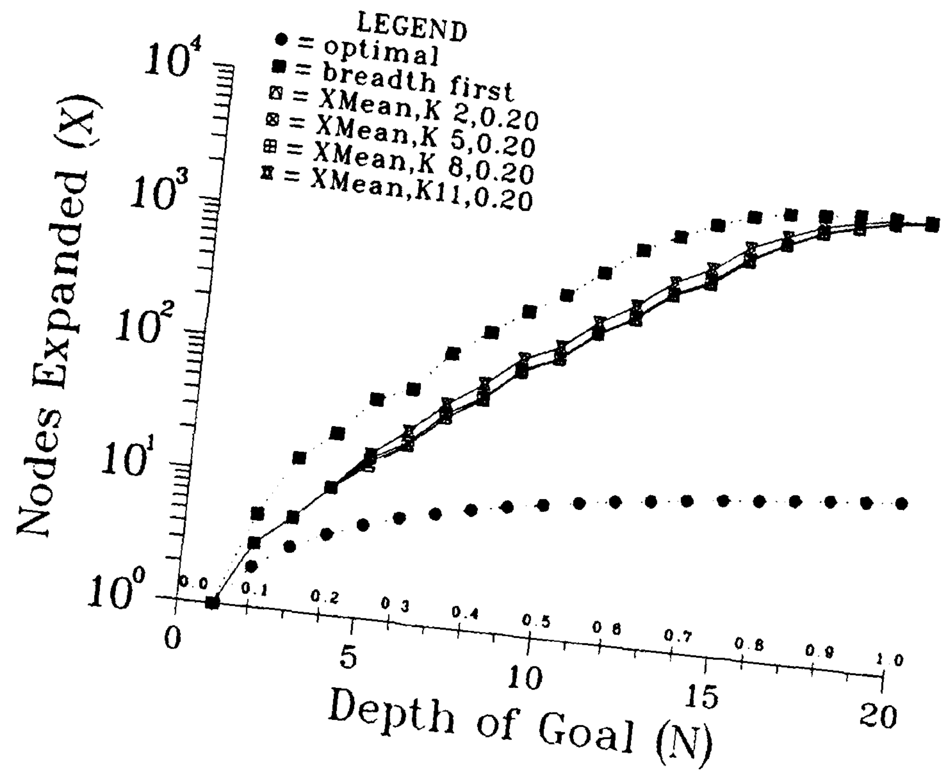


Figure 8.18b
Simulation Results
Heuristics K2, K5, K8, K11
Weight = 0.5
XMEAN

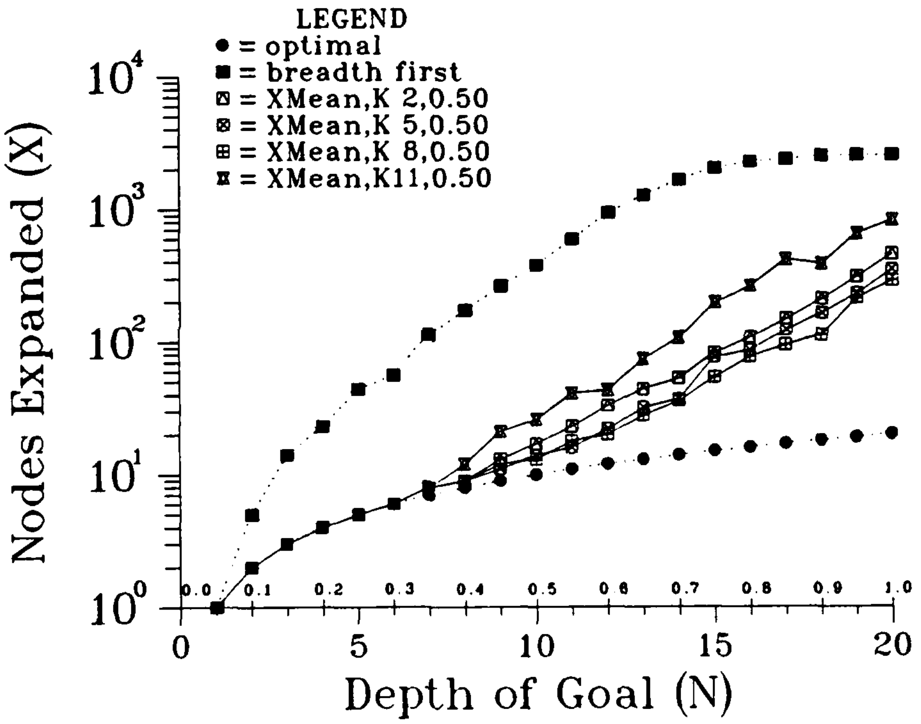


Figure 8.18c
Simulation Results
Heuristics K2, K5, K8, K11
Weight = 0.6
XMEAN

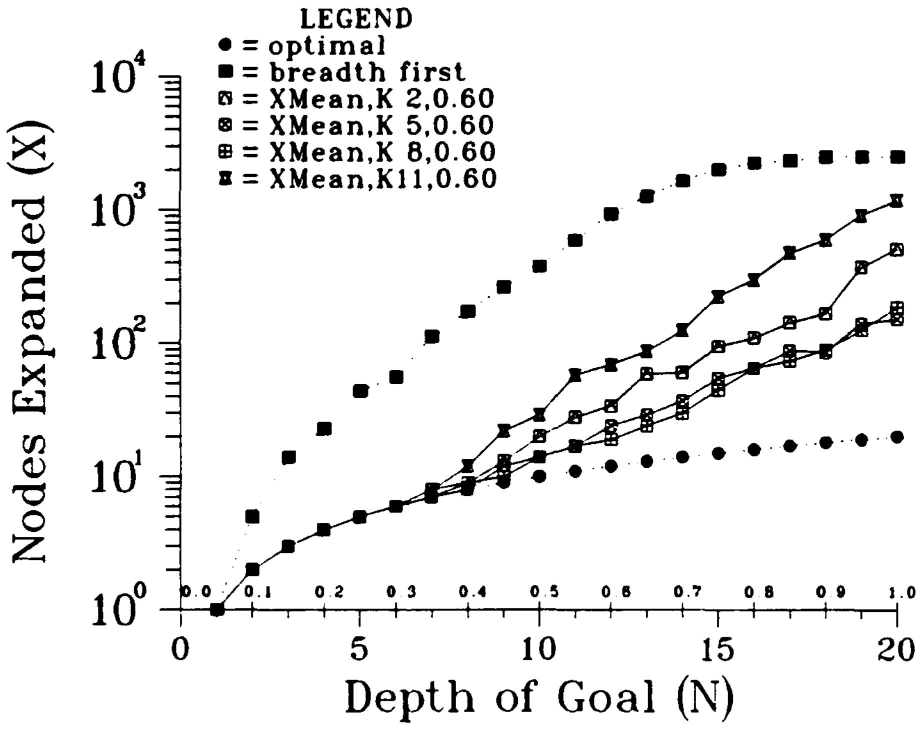


Figure 8.18d
Simulation Results
Heuristics K2, K5, K8, K11
Weight = 0.7
XMEAN

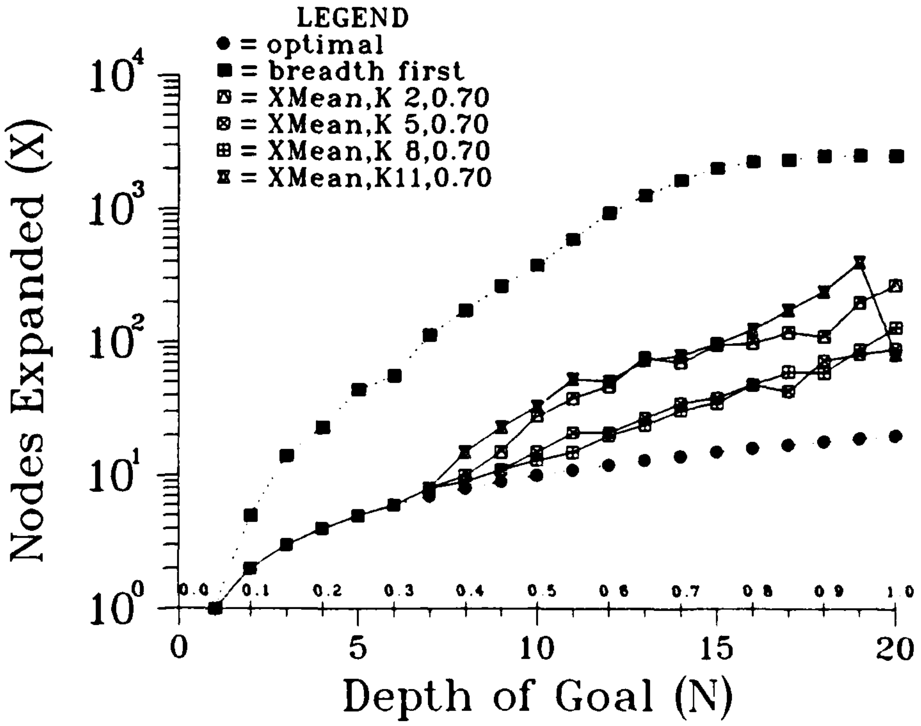


Figure 8.18e
Simulation Results
Heuristics K2, K5, K8, K11
Weight = 0.8
XMEAN

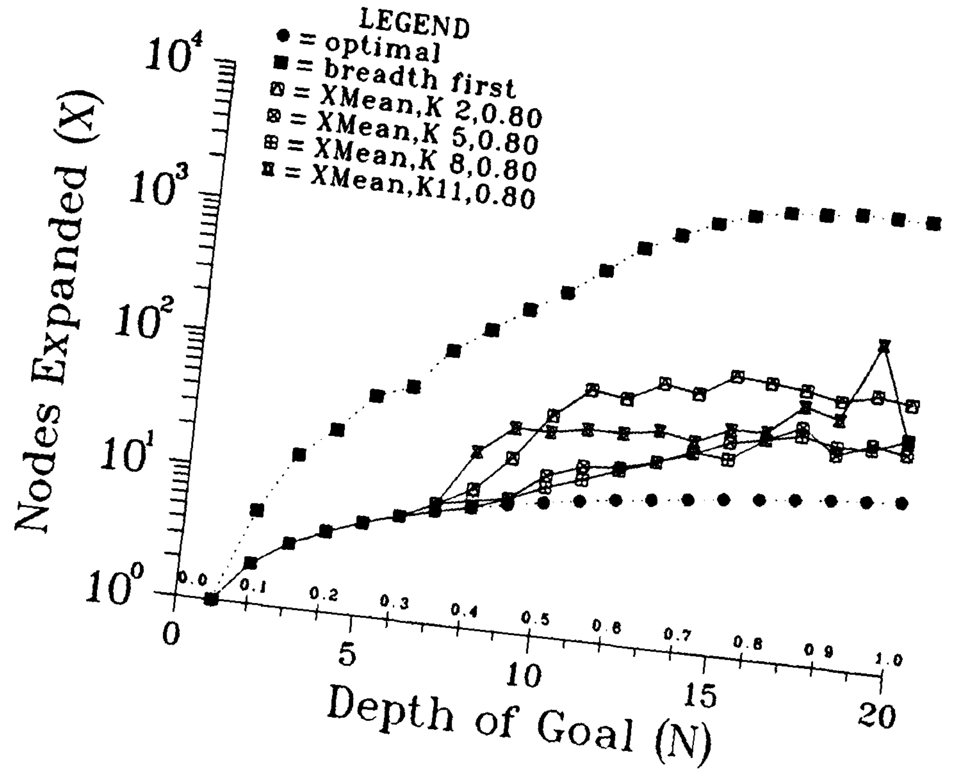


Figure 8.18f
Simulation Results
Heuristics K2, K5, K8, K11
Weight = 0.9
XMEAN

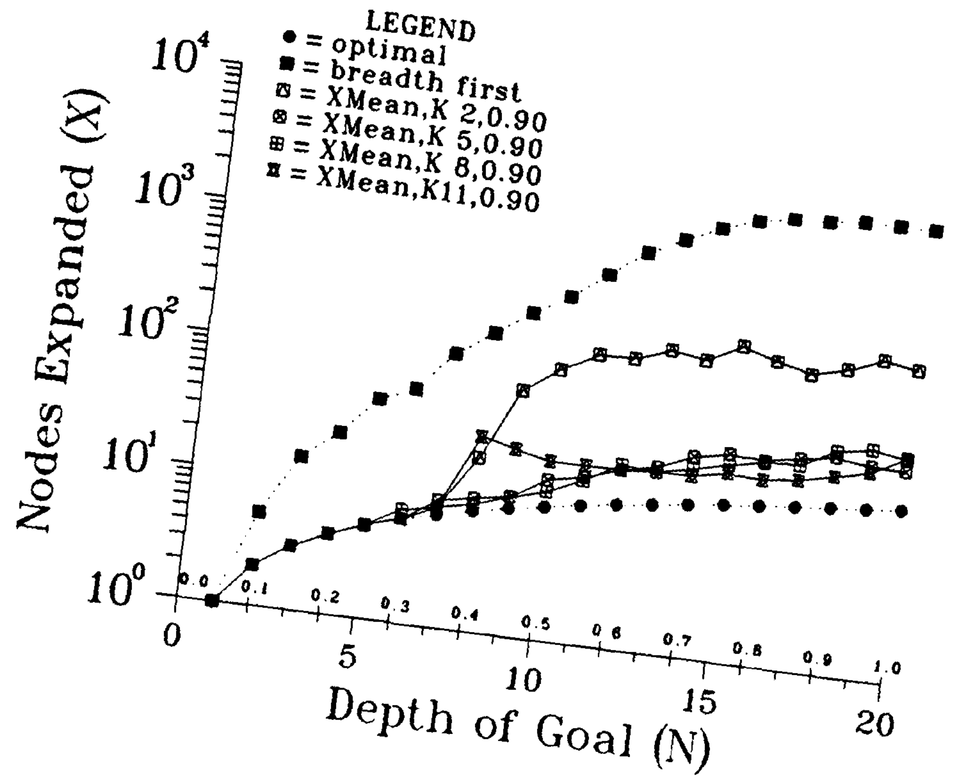


Figure 8.19a
Simulation Results
Heuristics K3, K6, K9, K12
Weight = 0.2
XMEAN

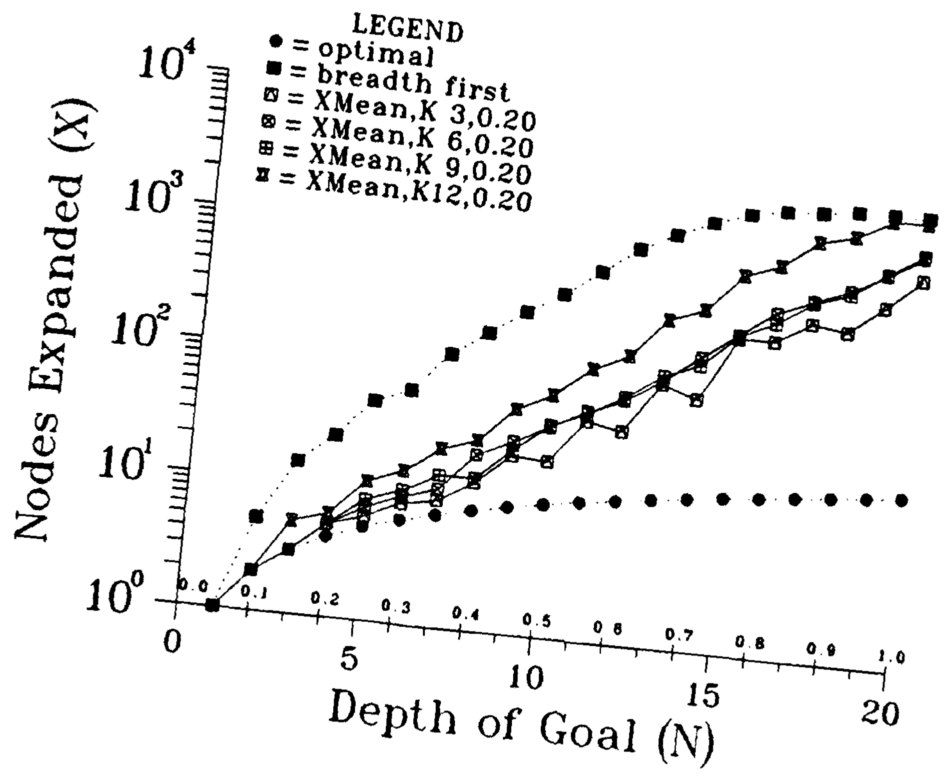


Figure 8.19b
Simulation Results
Heuristics K3, K6, K9, K12
Weight = 0.5
XMEAN

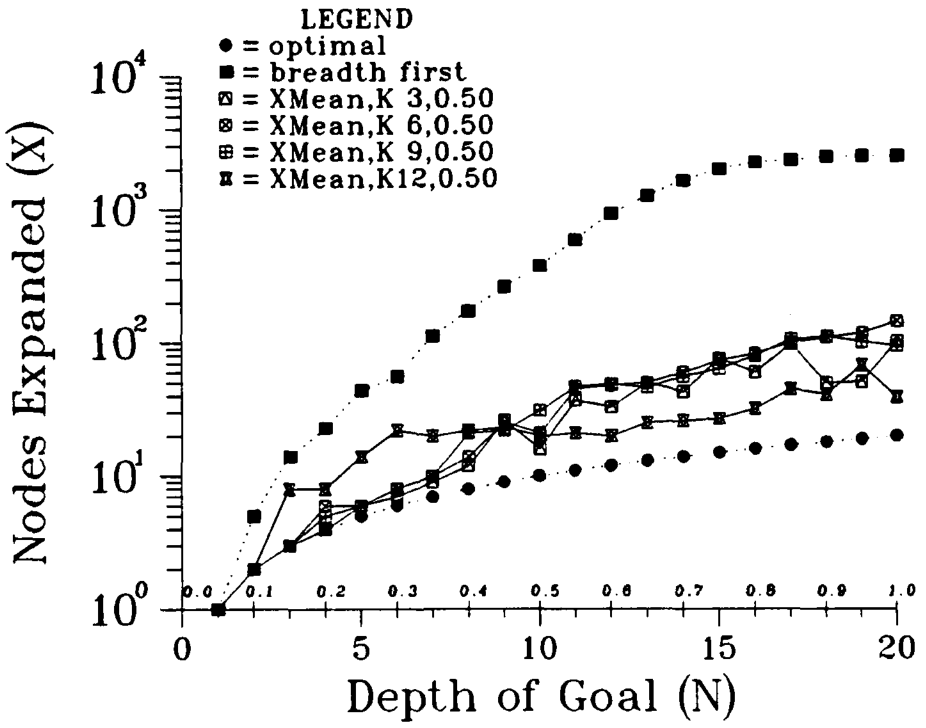


Figure 8.19c
Simulation Results
Heuristics K3, K6, K9, K12
Weight = 0.6
XMEAN

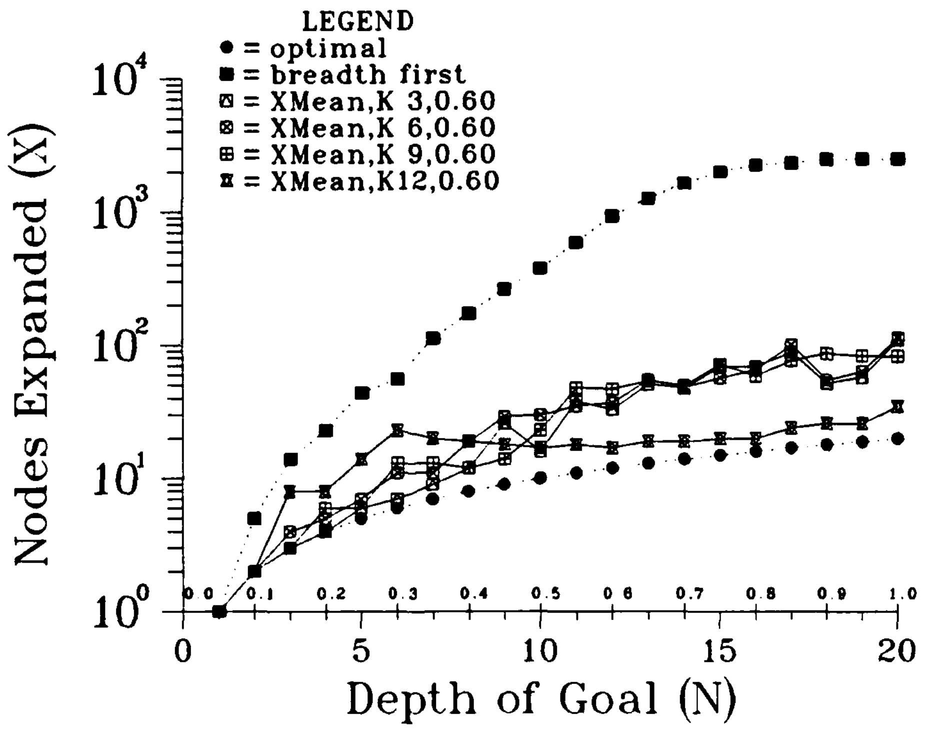


Figure 8.19d
Simulation Results
Heuristics K3, K6, K9, K12
Weight = 0.7
XMEAN

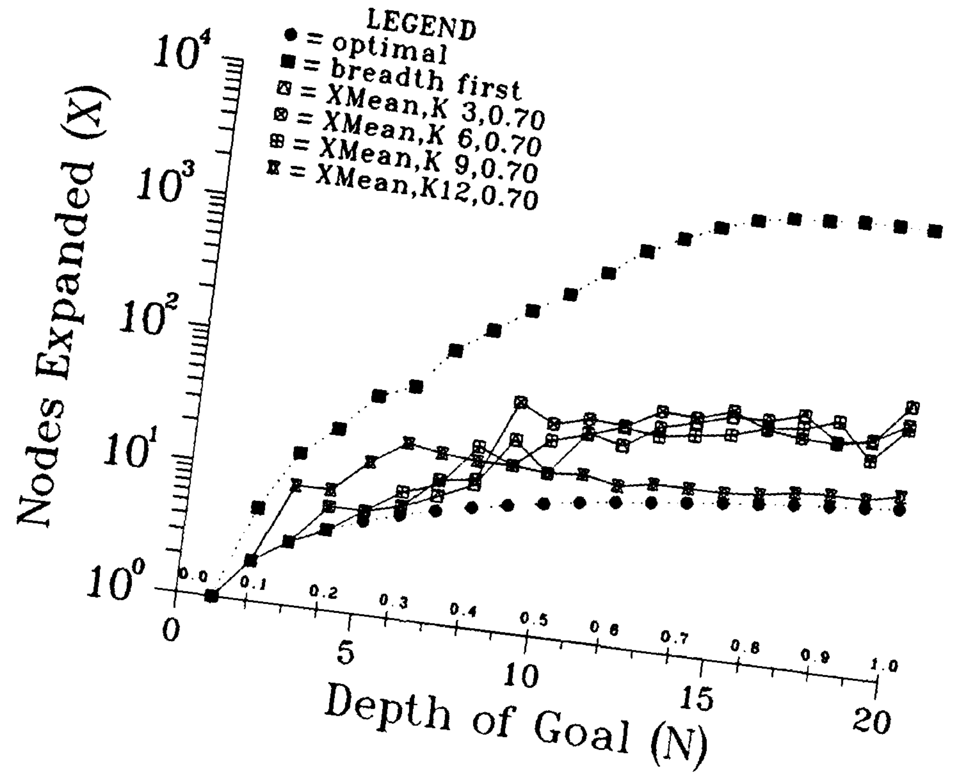


Figure 8.19e
Simulation Results
Heuristics K3, K6, K9, K12
Weight = 0.8
XMEAN

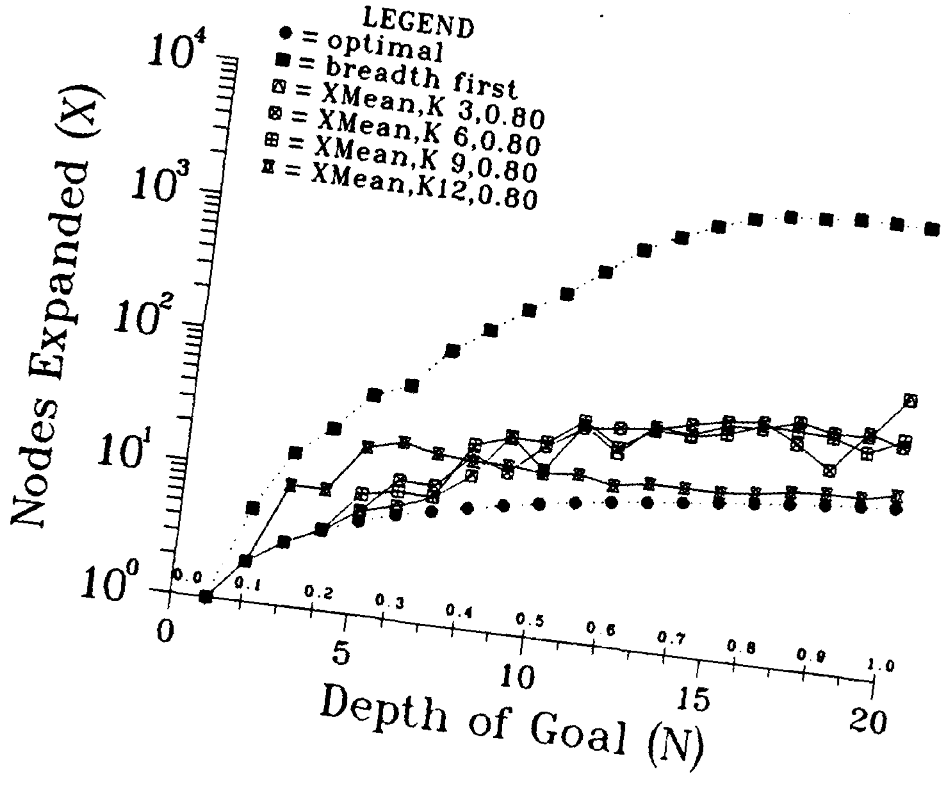


Figure 8.19f
Simulation Results
Heuristics K3, K6, K9, K12
Weight = 0.9
XMEAN

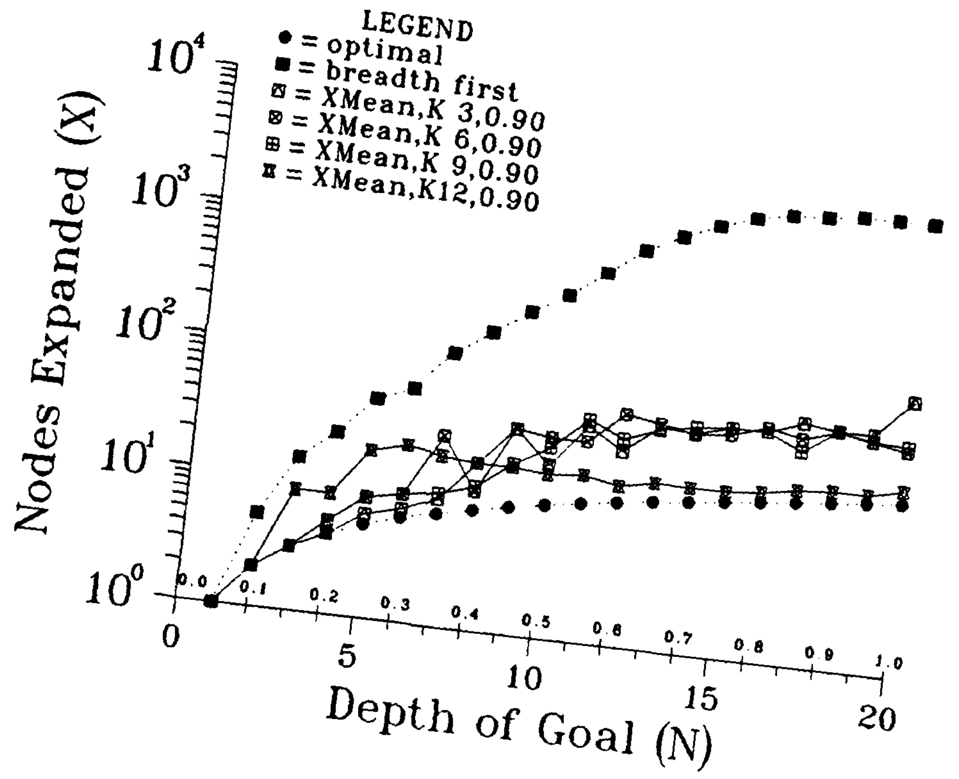


Figure 8.20
Simulation Results
Heuristics K1, K4, K7, K10
Weight = 0.9
LMEAN

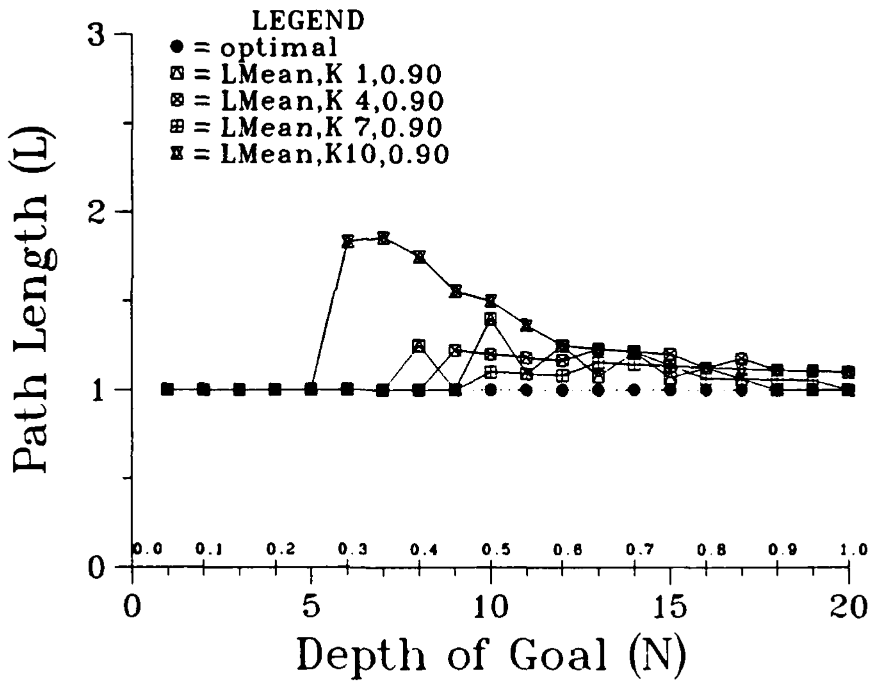


Figure 8.21
Simulation Results
Heuristics K2, K5, K8, K11
Weight = 0.9
LMEAN

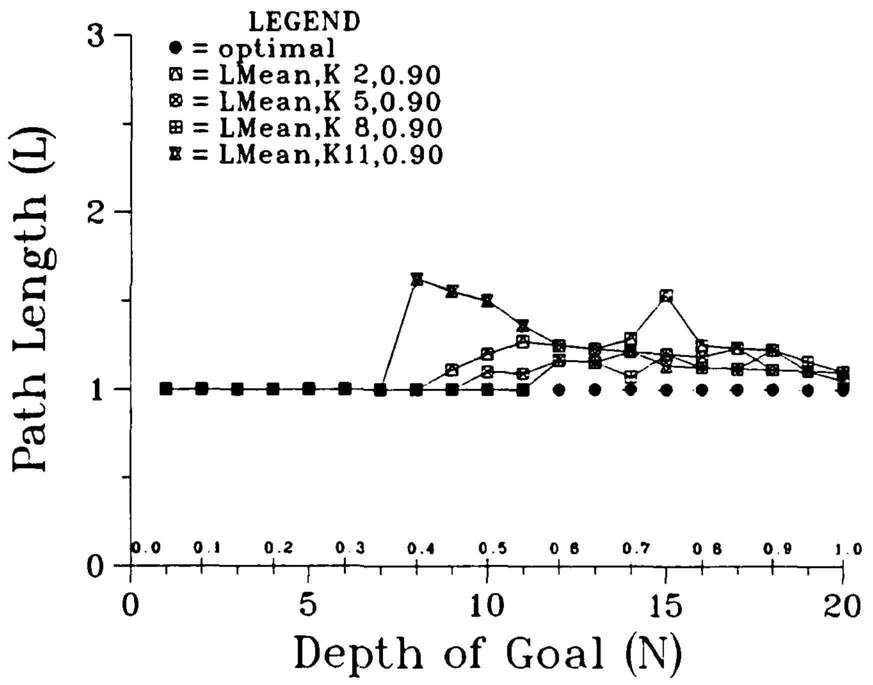
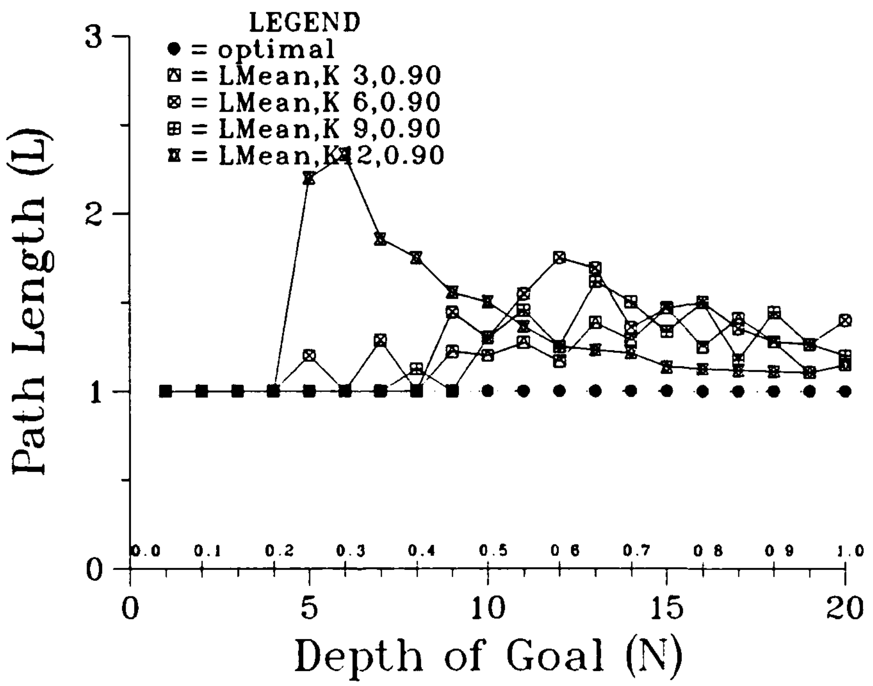


Figure 8.22
Simulation Results
Heuristics K3, K6, K9, K12
Weight = 0.9
LMEAN



E. CONCLUSIONS

Simulation by statistical profile provides interesting and varied behavior. There are four important contributing factors involved in achieving a good simulation: (1) Range, (2) Distribution, (3) Weight, and (4) Timing. Our choices of contrived heuristics provided a good filter for some of these items. Heuristics K1, K2, and K3 vary in Range; the simulation using the actual distribution (K7, K8, and K9) eliminated variations in distribution, and permitted focusing on effects of Timing; K10, K11, and K12 eliminated timing variations and focused exclusively on the effects of distribution.

Range is the distance at which the heuristic can see goal; it is inherent to the heuristic and cannot be altered. Profiling gives insight into what a heuristic's range is. Within range, simulation was excellent at all weights, assuming a fair distribution was used (meaning the contrived heuristic makes a serious attempt to reproduce the Source Profile. Outside a heuristic's range, the simulation is effective at low weights, but not effective at high weights.

Distribution is important to simulation, and has the advantage that it can be altered. For example, the Worst-Case distribution gave dramatically different results than K4-K9, and only the distribution was different. While K7-9 provided the most exact distribution possible from the Source Profiles, they didn't perform as closely to the actual heuristics as anticipated. What the distribution cannot capture is Timing. If the profiles could be augmented somehow to provide timing information, we expect that the simulation would be exact, regardless of the Range or Weight. Unfortunately, we know of no reasonable way to capture this level of detail.

Simulation works well at the lower weights for all of the contrived heuristics. It tends to soften the impact of poor timing and range, and to some extent, distribution by giving it less H component in the F-value.

The results show that heuristics cannot be considered equivalent purely on the basis of having identical KMIN and KMAX bounding functions. Even complete and exact profile duplication does not guarantee identical performance, or even close performance (as in K2 at W=0.8 and 0.9, Figures 8.18e-f) because the timing may differ dramatically. In addition to sharing the functions KMIN, KMEAN, and KMAX, it appears that having a small standard deviation reduces the variances caused by Timing.

Therefore, to ensure good simulation: (1) use heuristics with good range, tight standard deviation, and possessing a fair distribution; (2) for Heuristics without good range, lowering the weight will improve simulation; and finally, (3) if the standard deviation of the Source Profile is large, Timing will cause at least minor variations no matter the range of the heuristic (as in K3).

The technique of simulation is appealing and promising because the researcher can vary the distribution in order to focus on specific behavior. Our profiles were based on actual heuristics, but there is no reason they couldn't be represented as a table of values that the researcher could alter. Viewed in this manner, the researcher could contrive profiles and thereby control Range and standard deviation (which we could not do), in addition to distribution and weight (which we were limited to). Therefore, this mechanism appears useful in modelling specific heuristic behavior in controlled circumstances unlike those found in real life (such as a heuristic with linear error and no standard deviation), and empirically studying the results.

Besides modelling heuristics, future research could also be conducted into the equivalence of heuristics whose domains are different. If certain conditions permit a profile to simulate the actual heuristic keeping the domain fixed, if these conditions are preserved in another domain, does the profile retain its ability there? And, if profiles can be used to esablish the equivalence of heuristics in the same domain, are there conditions under which the equality of profiles from distinct domains establish equivalence also? That is, if a heuristic in the 6-puzzle has the same profile as a heuristic in the checkerboard domain, can they be called equivalent? Or, can a heuristic be moved "in spirit" by its statistical profile to another domain and still have power?

Pearl (1984, Chapters 6 and 7) investigated the behavior of UA* on the assumption that H(n) is a random variable whose distribution depends only upon G*(n) (which is the actual minimal distance from the root to the node n, and closely related to what we have called G), and H*(n) (what we have called i, or the actual distance remaining to the goal), and that H(n) is independant for each node. Our study sheds insight into the plausibility of Pearl's assumptions, permitting the general search problem to be viewed as consisting of three independant variable components: (1) the search algorithm (such as A*, Weighted A*, etc.), (2) the graph or domain (such as the 6-Puzzle, 8-Puzzle, checkerboard, etc.), and (3) the heuristic (a random variable). If random variables can be used under the proper conditions to simulate heuristics, as we have shown, then we can vary any of the three above independently of the others and thereby attempt to get real insight into the general searching process.

Using our tools and techniques, a world of research possiblilties have been opened up, and we hope that they will be beneficial when applied to answering these issues.

APPENDICES

```
***********************************************************************
*                                                                     *
*                             APPENDIX A                              *
*                                                                     *
*                  General Beads World Tools Modules                  *
*                                                                     *
***********************************************************************
```

```
(*
 *       Utilities Module  (v4.0  23-Feb-86  AJC/SRH)
 *)

module utilities (input, output);
    const
        max_positions = 10;
        no_puzzle = '          ';
        center = 1;
        first = 2;

    type
        positions = center..max_positions;
        puzzle_state = packed array [positions] of char;

        node_ptr = ^puzzle_node;

        neighbor_node_ptr = ^neighbor_node;
        neighbor_node = record
                neighbor : node_ptr;
                next : neighbor_node_ptr;
                end;

        puzzle_node = record
                state : puzzle_state;
                left, right, sort_left, sort_right : node_ptr;
                neighbors : neighbor_node_ptr;
                parent : node_ptr;
                g_value, h_value : integer;
                f_value : real;
                end;

(*
 * puzzle I/O functions
 *)

    [global]
    procedure read_state (var s : puzzle_state; n : integer);
        var
            i, p : integer;
            ch : char;
        begin
        repeat
            read (ch);
        until ch = '(';
        for i := center to n do
            begin
            read (p);
            s[i] := chr (p);
            end;
        repeat
            read (ch);
        until (ch = ')') or eoln;
        if ch <> ')' then
            begin
            writeln ('  *** Error - Improper format on state input.');
            halt;
            end;
        end;

    [global]
    procedure print_state (s : puzzle_state; n : integer);
        var
            i : integer;
        begin
        write (' (');
        for i := center to n do
            write (ord(s[i]):2,' ');
        write (') ');
        end;

(*
 * node and list primitives
 *)

    [global]
    function create_puzzle_node (s : puzzle_state): node_ptr;
        var
            n : node_ptr;
        begin
        new (n);
        n^.state := s;
        n^.left := nil;
        n^.right := nil;
        n^.sort_left := nil;
        n^.sort_right := nil;
        n^.neighbors := nil;
        n^.parent := nil;
        n^.g_value := 0;
        n^.h_value := 0;
        n^.f_value := 0.0;
        create_puzzle_node := n;
        end;

    [global]
    procedure free_node (var n : node_ptr);
        var
        m, p : neighbor_node_ptr;
    begin
    m := n^.neighbors;
    while m <> nil do
        begin
        p := m;
        m := m^.next;
        dispose (p);
        end;
    dispose (n);
    end;

[global]
procedure create_empty_list (var list : node_ptr);
    begin
    list := create_puzzle_node (no_puzzle);
    list^.left := list;
    list^.right := list;
    list^.f_value := 0.0;
    list^.g_value := 0;
    end;

[global]
function is_empty (list : node_ptr) : boolean;
    begin
    is_empty := (list^.g_value = 0);
    end;

[global]
procedure place_on_end_of_list (p, list : node_ptr);
    begin
    p^.left := list^.left;
    p^.right := list;
    list^.left^.right := p;
    list^.left := p;
    list^.g_value := list^.g_value + 1;
    end;

[global]
procedure place_in_ascending_order (p, list : node_ptr);
    var
        q, r : node_ptr;
    begin
    q := list^.right;
    while (q <> list) and (p^.f_value > q^.f_value) do
        q := q^.right;
    r := q^.left;
    r^.right := p;
    q^.left := p;
    p^.left := r;
    p^.right := q;
    list^.g_value := list^.g_value + 1;
    end;
    [global]
    function remove_from_front_of_list (list : node_ptr) : node_ptr;
        var
            p, q : node_ptr;
        begin
        if list^.g_value = 0 then
            remove_from_front_of_list := nil
        else
            begin
            p := list^.right;
            q := p^.right;
            q^.left := list;
            list^.right := q;
            p^.left := nil;
            p^.right := nil;
            list^.g_value := list^.g_value - 1;
            remove_from_front_of_list := p;
            end;
        end;

    [global]
    procedure delete_from_list (p, list : node_ptr);
        var
            l, r : node_ptr;
        begin
        if p <> nil then
            begin
            l := p^.left;
            r := p^.right;
            l^.right := r;
            r^.left := l;
            p^.left := nil;
            p^.right := nil;
            list^.g_value := list^.g_value - 1;
            end;
        end;

    [global]
    procedure free_list (var list : node_ptr);
        var
            p : node_ptr;
        begin
        while list^.g_value > 0 do
            begin
            p := remove_from_front_of_list (list);
            free_node (p);
            end;
        free_node (list);
        end;

(*
 * search tree primitives
```

```
*)

    [global]
    procedure insert_in_tree (n : node_ptr; var tree : node_ptr);
        var
            q, r : node_ptr;
        begin
        if tree = nil then
            tree := n
        else
            begin
            q := tree;
            while q <> nil do
                begin
                r := q;
                if n^.state < q^.state then
                    q := q^.sort_left
                else
                    q := q^.sort_right;
                end;
            if n^.state < r^.state then
                r^.sort_left := n
            else
                r^.sort_right := n;
            end;
        end;

    [global]
    function find_in_tree (n, tree : node_ptr) : node_ptr;
        var
            p : node_ptr;
            found : boolean;
        begin
        found := false;
        p := tree;
        while (p <> nil) and (not found) do
            begin
            if p^.state = n^.state then
                found := true
            else
                if n^.state < p^.state then
                    p := p^.sort_left
                else
                    p := p^.sort_right;
            end;
        if found then
            find_in_tree := p
        else
            find_in_tree := nil;
        end;

    [global]
    function find_state_in_tree (s : puzzle_state;
                              tree : node_ptr) : node_ptr;
    var
        p : node_ptr;
        found : boolean;
    begin
    found := false;
    p := tree;
    while (p <> nil) and (not found) do
        begin
        if p^.state = s then
            found := true
        else
            if s < p^.state then
                p := p^.sort_left
            else
                p := p^.sort_right;
        end;
    if found then
        find_state_in_tree := p
    else
        find_state_in_tree := nil;
    end;

[global]
procedure free_binary_tree (var t : node_ptr);
    var
        p, q : node_ptr;
    begin
    if t <> nil then
        begin
        free_binary_tree (t^.sort_left);
        free_binary_tree (t^.sort_right);
        free_node (t);
        t := nil;
        end;
    end;

[global]
procedure free_graph (var g : node_ptr);
    var
        n : neighbor_node_ptr;
    begin
    if (g^.state <> no_puzzle) then
        begin
        g^.state := no_puzzle;
        n := g^.neighbors;
        while n <> nil do
            begin
            free_graph (n^.neighbor);
            n := n^.next;
            end;
        free_node (g);
        end;
```

```
            end;

        end.
```

```
(*
 *       Control Module  (v4.0  23-Feb-86  AJC/SRH)
 *)

module control (input, output);
    const
(*
 *  From UTILITIES Import CONST
 *)
        max_positions = 10;
        no_puzzle = '          ';
        center = 1;
        first = 2;

(*
 *  CONTROL
 *)
        max_links = max_positions;
        max_levels = 99;

    type
(*
 *  From UTILITIES Import TYPE
 *)
        positions = center..max_positions;
        puzzle_state = packed array [positions] of char;

        node_ptr = ^puzzle_node;

        neighbor_node_ptr = ^neighbor_node;
        neighbor_node = record
                neighbor : node_ptr;
                next : neighbor_node_ptr;
                end;

        puzzle_node = record
                state : puzzle_state;
                left, right, sort_left, sort_right : node_ptr;
                neighbors : neighbor_node_ptr;
                parent : node_ptr;
                g_value, h_value : integer;
                f_value : real;
                end;

(*
 *  CONTROL
 *)
        link_array = array[first..max_positions] of boolean;

        level_record = record
                count : integer;
                list : node_ptr;
                end;

        level_array = array [0..max_levels] of level_record;

        graph_descriptor = record
                depth, generated, expanded : integer;
                level : level_array;
                end;

        results_descriptor = record
                solved : boolean;
                path_length, min_path_length, generated,
                    expanded, heuristic : integer;
                weight : real;
                start, goal : node_ptr;
                end;

(*
 * puzzle characteristics
 *)

    var
        num_positions, num_links : [global] integer;
        link : [global] link_array;

(*
 *  From UTILITIES Import:
 *)

    [external]
    procedure print_state (s : puzzle_state; n : integer);
        external;

    [external]
    function create_puzzle_node (p : puzzle_state): node_ptr;
        external;

    [external]
    procedure free_node (var n : node_ptr);
        external;

    [external]
    procedure create_empty_list (var list : node_ptr);
        external;

    [external]
    function is_empty (list : node_ptr) : boolean;
        external;

    [external]
    procedure place_on_end_of_list (p, list : node_ptr);
        external;
    [external]
    procedure place_in_ascending_order (p, list : node_ptr);
        external;

    [external]
    function remove_from_front_of_list (list : node_ptr) : node_ptr;
        external;

    [external]
    procedure delete_from_list (p, list : node_ptr);
        external;

    [external]
    procedure free_list (var list : node_ptr);
        external;

    [external]
    procedure insert_in_tree (n : node_ptr; var tree : node_ptr);
        external;

    [external]
    function find_in_tree (n, tree : node_ptr) : node_ptr;
        external;

(*
 *  From HEURISTIC Import:
 *)

    [external]
    procedure initialize_heuristics;
        external;

    [external]
    function estimated_distance (heuristic : integer;
                                 current   : node_ptr;
                                 goal      : puzzle_state;
                                 c_star    : integer) : integer;
        external;

(*
 *  CONTROL
 *)

    [global]
    procedure initialize_control (np, nl : integer; l : link_array);
        var
            i : positions;
        begin
        num_positions := np;
        num_links := nl;
        for i := first to np do
            link[i] := l[i];
    initialize_heuristics;
    end;

procedure generate_successors (p : puzzle_state;
                               successor_list : node_ptr);
    var
        i : integer;
        blnk : char;

    procedure add_successor (source, dest : integer);
        var
            c : node_ptr;
        begin
        c := create_puzzle_node (p);
        c^.state[dest] := c^.state[source];
        c^.state[source] := blnk;
        place_on_end_of_list (c, successor_list);
        end;

    begin
    blnk := chr(0);
    for i := first to num_positions do
        begin
        if p[i] <> blnk then
            begin
            if (i = num_positions) and (p[first] = blnk) then
                add_successor (num_positions, first);
            if (i <> num_positions) and (p[i+1] = blnk) then
                add_successor (i, i+1);
            if (i = first) and (p[num_positions] = blnk) then
                add_successor (first, num_positions);
            if (i <> first) and (p[i-1] = blnk) then
                add_successor (i, i-1);
            if (p[center] = blnk) and link[i] then
                add_successor (i, center);
            end;
        end;
    if p[center] <> blnk then
        begin
        for i := first to num_positions do
            if link[i] and (p[i] = blnk) then
                add_successor (center, i);
        end;
    end;

procedure add_neighbor (neighbor, current : node_ptr);
    var
        n : neighbor_node_ptr;
    begin
    new (n);
    n^.neighbor := neighbor;
    n^.next := current^.neighbors;
    current^.neighbors := n;
```

```
        end;

(*
 *  Search tree generation
 *)

    [global]
    procedure initialize_graph_descriptor (var g : graph_descriptor);
        var
            i : integer;
        begin
        g.depth := 0;
        g.generated := 0;
        g.expanded := 0;
        for i := 0 to max_levels do
            begin
            g.level[i].count := 0;
            create_empty_list (g.level[i].list);
            end;
        end;

    [global]
    procedure generate_graph (start_state : puzzle_state;
                              var search_tree, graph : node_ptr;
                              var g : graph_descriptor);
        var
            start, current, c, p : node_ptr;
            open, successor_list : node_ptr;
            depth, nodes_generated, nodes_expanded : integer;

        begin
        create_empty_list (open);
        create_empty_list (successor_list);
        search_tree := nil;
        graph := nil;

        nodes_generated := 1;
        nodes_expanded := 0;

        start := create_puzzle_node (start_state);
        start^.g_value := 0;
        start^.f_value := 0.0;
        insert_in_tree (start, search_tree);
        graph := start;
        place_in_ascending_order (start, open);

        while not is_empty (open) do
            begin
            current := remove_from_front_of_list (open);
            nodes_expanded := nodes_expanded + 1;
            depth := current^.g_value;
            g.level[depth].count := g.level[depth].count + 1;
            place_on_end_of_list (current, g.level[depth].list);
            generate_successors (current^.state, successor_list);
            while not is_empty (successor_list) do
                begin
                c := remove_from_front_of_list (successor_list);
                c^.g_value := depth + 1;
                c^.f_value := depth + 1;
                p := find_in_tree (c, search_tree);
                if p = nil then
                    begin
                    insert_in_tree (c, search_tree);
                    add_neighbor (c, current);
                    place_in_ascending_order (c, open);
                    end
                else
                    begin
                    add_neighbor (p, current);
                    free_node (c);
                    end;
                nodes_generated := nodes_generated + 1;
                end;
            end;

        g.depth := depth;
        g.generated := nodes_generated;
        g.expanded := nodes_expanded;
        free_list (open);
        free_list (successor_list);
        end;

(*
 *  Problem solution routines
 *)

    [global]
    procedure initialize_results (var r : results_descriptor);
        begin
        r.heuristic := 0;
        r.weight := 0.0;
        r.generated := 0;
        r.expanded := 0;
        r.path_length := 0;
        r.min_path_length := 0;
        r.start := nil;
        r.goal := nil;
        end;

    [global]
    procedure print_puzzle_solution (r : results_descriptor);

        procedure pps (n : node_ptr);
            begin
            if n^.state <> r.start^.state then
                pps (n^.parent);
            print_state (n^.state, num_positions);
            writeln;
            end;

        begin
        pps (r.goal)
        end;


    [global]
    procedure ordered_search (start, goal : puzzle_state;
                              heuristic : integer;
                              weight : real;
                              var results : results_descriptor);
        var
            open, successor_list, search_tree : node_ptr;
            start_node, current, p, c : node_ptr;
            nodes_generated, nodes_expanded : integer;

        begin
        create_empty_list (open);
        create_empty_list (successor_list);
        search_tree := nil;

        nodes_generated := 1;
        nodes_expanded := 0;

        start_node := create_puzzle_node (start);
        start_node^.g_value := 0;
        start_node^.h_value := estimated_distance (heuristic, start_node,
                                      goal, results.min_path_length);
        start_node^.f_value := (1.0 - weight) * start_node^.g_value +
                                    weight * start_node^.h_value;
        insert_in_tree (start_node, search_tree);
        place_in_ascending_order (start_node, open);

        repeat
            current := remove_from_front_of_list (open);
            if (current^.state <> goal) then
                begin
                nodes_expanded := nodes_expanded + 1;
                generate_successors (current^.state, successor_list);
                while not is_empty (successor_list) do
                    begin
                    c := remove_from_front_of_list (successor_list);
                    c^.g_value := current^.g_value + 1;
                    p := find_in_tree (c, search_tree);
                    if p = nil then
                        begin
                        c^.parent := current;
                        c^.h_value := estimated_distance (heuristic, c,
                                          goal, results.min_path_length);
                        c^.f_value := (1.0 - weight) * c^.g_value +
                                   weight * c^.h_value;
                    insert_in_tree (c, search_tree);
                    place_in_ascending_order (c, open);
                    end
                else
                    begin
                    if c^.g_value < p^.g_value then
                        begin
                        p^.parent := current;
                        p^.g_value := c^.g_value;
                        p^.f_value := (1.0 - weight) * p^.g_value +
                                          weight * p^.h_value;
                        if not ((p^.left = nil) and
                                (p^.right = nil)) then
                            delete_from_list (p, open);
                        place_in_ascending_order (p, open);
                        end;
                    free_node (c);
                    end;
                nodes_generated := nodes_generated + 1;
                end;
            end;

    until is_empty (open) or (current^.state = goal);

    if current^.state = goal then
        results.solved := true
    else
        results.solved := false;
    results.start := start_node;
    results.goal := current;
    results.heuristic := heuristic;
    results.weight := weight;
    results.path_length := current^.g_value;
    results.generated := nodes_generated;
    results.expanded := nodes_expanded;
    free_list (successor_list);
    free_node (open);
    end;


[global]
procedure graph_search (start, goal : puzzle_state;
                        heuristic : integer;
                        weight : real;
                        var results : results_descriptor);
    var
        open, successor_list, search_tree : node_ptr;
        start_node, current, p, c : node_ptr;
        nodes_generated, nodes_expanded : integer;

    procedure update (p : node_ptr);
        var
        n : neighbor_node_ptr;
        m : node_ptr;
    begin
    n := p^.neighbors;
    while n <> nil do
        begin
        m := n^.neighbor;
        if (p^.g_value + 1) < m^.g_value then
            begin
            m^.g_value := p^.g_value + 1;
            m^.f_value := (1.0 - weight) * m^.g_value +
                                    weight * m^.h_value;
            if not ((m^.left = nil) and (m^.right = nil)) then
                begin
                delete_from_list (m, open);
                place_in_ascending_order (m, open);
                end;
             update (m);
            end;
        n := n^.next;
        end;
    end;

begin
create_empty_list(open);
create_empty_list(successor_list);
search_tree := nil;

nodes_generated := 1;
nodes_expanded := 0;

start_node := create_puzzle_node (start);
start_node^.g_value := 0;
start_node^.h_value := estimated_distance (heuristic, start_node,
                            goal, results.min_path_length);
start_node^.f_value := (1.0 - weight) * start_node^.g_value +
                               weight * start_node^.h_value;
insert_in_tree (start_node, search_tree);
place_in_ascending_order (start_node, open);

repeat
    current := remove_from_front_of_list (open);
    if (current^.state <> goal) then
        begin
        nodes_expanded := nodes_expanded + 1;
        generate_successors (current^.state, successor_list);
        while not is_empty (successor_list) do
            begin
            c := remove_from_front_of_list (successor_list);
            c^.g_value := current^.g_value + 1;
            p := find_in_tree (c, search_tree);
            if p = nil then
                begin
```

```
                        c^.parent := current;
                        c^.h_value := estimated_distance (heuristic, c,
                                        goal, results.min_path_length);
                        c^.f_value := (1.0 - weight) * c^.g_value +
                                        weight * c^.h_value;
                        add_neighbor (c, current);
                        insert_in_tree (c, search_tree);
                        place_in_ascending_order (c, open);
                        end
                    else
                        begin
                        add_neighbor (p, current);
                        if c^.g_value < p^.g_value then
                            begin
                            p^.g_value := c^.g_value;
                            p^.f_value := (1.0 - weight) * p^.g_value +
                                            weight * p^.h_value;
                            p^.parent := current;
                            if ((p^.left = nil) and
                                (p^.right = nil)) then
                                update (p)
                            else
                                begin
                                delete_from_list (p, open);
                                place_in_ascending_order (p, open);
                                end;
                            end;
                        free_node (c);
                        end;
                    nodes_generated := nodes_generated + 1;
                    end;
                end;

    until is_empty (open) or (current^.state = goal);

    if current^.state = goal then
        results.solved := true
    else
        results.solved := false;
    results.start := start_node;
    results.goal := current;
    results.heuristic := heuristic;
    results.weight := weight;
    results.path_length := current^.g_value;
    results.generated := nodes_generated;
    results.expanded := nodes_expanded;
    free_list (successor_list);
    free_node (open);
    end;

end.
```

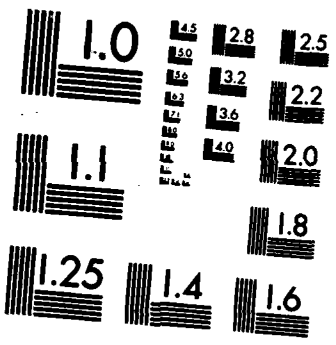
```
(*
 *        Heuristics Module  (v4.1  23-Feb-86  AJC/SRH)
 *)

module heuristic (input, output, profile_input, profile_output);
    const
(*
 *  From UTILITIES Import CONST
 *)
        max_positions = 10;
        no_puzzle = '          ';
        center = 1;
        first = 2;
(*
 *  From CONTROL Import CONST
 *)
        max_links = max_positions;
        max_levels = 99;
(*
 *  From STATISTIC Import CONST
 *)
        max_pairs = 100;
(*
 *  HEURISTIC
 *)
        max_heuristics = 24;
        max_n = 20;
        max_k = 75;
        max_name = 10;
        max_file_name = 30;

    type
(*
 *  From UTILITIES Import TYPE
 *)
        positions = center..max_positions;
        puzzle_state = packed array [positions] of char;

        node_ptr = ^puzzle_node;

        neighbor_node_ptr = ^neighbor_node;
        neighbor_node = record
                neighbor : node_ptr;
                next : neighbor_node_ptr;
                end;

        puzzle_node = record
                state : puzzle_state;
                left, right, sort_left, sort_right : node_ptr;
                neighbors : neighbor_node_ptr;
                parent : node_ptr;
                g_value, h_value : integer;
                f_value : real;
                end;
(*
 *  From CONTROL Import TYPE
 *)
         link_array = array [first..max_links] of boolean;

         level_record = record
                count : integer;
                list : node_ptr;
                end;

         level_array = array [0..max_levels] of level_record;

         graph_descriptor = record
                depth, generated, expanded : integer;
                level : level_array;
                end;

(*
 *  From STATISTIC Import TYPE
 *)
         distribution_index = 1..max_pairs;
         distribution_type = (normal, linear, nonlinear);

         distribution_pointer = ^distribution_record;
         distribution_record = record
                name : distribution_type;
                pairs : distribution_index;
                abscissa, ordinate : array [distribution_index] of real;
                end;

(*
 *  HEURISTIC
 *)
         name_string = packed array [1..max_name] of char;
         file_name_string = packed array [1..max_file_name] of char;

         profile_pointer = ^profile_record;
         profile_record = record
                name : name_string;
                heuristic : 0..max_heuristics;
                min, max, count : array [0..max_n] of integer;
                mean, stdev : array [0..max_n] of real;
                histogram : array [0..max_n, 0..max_k] of integer;
                end;

    var
(*
 *  From CONTROL Import VAR
 *)
         link : [external] link_array;
```

```
        num_positions : [external] integer;

(*
 * Local
 *)
        start : puzzle_state;
        inv_search_tree, inv_graph : node_ptr;
        inv_g : graph_descriptor;

        profile_input, profile_output : text;
        frequency : array [1..max_heuristics, 0..max_n, 0..max_k]
                          of integer;
        profile : array [1..max_heuristics] of profile_pointer;

(*
 * From UTILITIES Import:
 *)

    [external]
    function create_puzzle_node (s : puzzle_state) : node_ptr;
        external;

    [external]
    procedure free_binary_tree (var t : node_ptr);
        external;

    [external]
    function find_state_in_tree (s : puzzle_state;
                                 tree : node_ptr) : node_ptr;
        external;

(*
 * From CONTROL Import:
 *)

    [external]
    procedure initialize_graph_descriptor (var g : graph_descriptor);
        external;

    [external]
    procedure generate_graph (start : puzzle_state;
                              var search_tree, graph : node_ptr;
                              var g : graph_descriptor);
        external;

(*
 * From STATISTIC Import:
 *)

    [external]
    procedure initialize_statistics;
        external;
    [external]
    function random_integer_between (m, n : integer) : integer;
        external;

    [external]
    function random_by_distribution (d : distribution_type) : real;
        external;

(*
 *  HEURISTIC
 *)

    procedure initialize_input_profiles;
        var
            h : integer;
        begin
        for h := 1 to max_heuristics do
            profile[h] := nil;
        end;

    [global]
    procedure create_profile (var p : profile_pointer);
        var
            i, j : integer;
        begin
        new (p);
        with p^ do
            begin
            for i := 1 to max_name do
                name[i] := ' ';
            heuristic := 0;
            for i := 1 to max_n do
                begin
                min[i] := max_k;
                max[i] := 0;
                count[i] := 0;
                mean[i] := 0.0;
                stdev[i] := 0.0;
                for j := 1 to max_k do
                    histogram[i, j] := 0;
                end;
            end;
        end;

    [global]
    procedure read_profiles (file_name : file_name_string);
        var
            p : profile_pointer;
            name : name_string;
            number_of_entries, frequency : integer;
            nsum, ksum, k2sum, percent : real;
            i, k, n : integer;
        begin
    open (profile_input, file_name, history := old);
    reset (profile_input);
    while not eof (profile_input) do
        begin
        create_profile (p);
        with p^ do
            begin
            readln (profile_input, name, heuristic,
                    number_of_entries);
            for i := 1 to number_of_entries do
                begin
                readln (profile_input, n, k, percent, frequency);
                histogram[n, k] := frequency;
                end;
            for n := 0 to max_n do
                begin
                nsum := 0.0;
                ksum := 0.0;
                k2sum := 0.0;
                for k := 0 to max_k do
                    if (histogram [n, k] <> 0) then
                        begin
                        nsum := nsum + histogram [n, k];
                        ksum := ksum + k * histogram [n, k];
                        k2sum := k2sum + k * k * histogram [n, k];
                        if (k < min[n]) then
                            min[n] := k;
                        if (k > max[n]) then
                            max[n] := k;
                        count[n] := count[n] + histogram [n, k];
                        end;
                mean[n] := ksum / nsum;
                stdev[n] := sqrt (abs (ksum * ksum / nsum - k2sum)
                                   / (nsum - 1));
                end;
            end;
        profile[p^.heuristic] := p;
        end;
    close (profile_input);
    end;

procedure initialize_output_profiles;
    var
        h, n, k : integer;
    begin
    for h := 1 to max_heuristics do
        for n := 0 to max_n do
            for k := 0 to max_k do
                frequency[h, n, k] := 0;
    end;

[global]
procedure initialize_heuristics;
```

```
        begin
        initialize_statistics;
        initialize_graph_descriptor (inv_g);
        inv_graph := create_puzzle_node (no_puzzle);;
        inv_search_tree := inv_graph;
        initialize_input_profiles;
        initialize_output_profiles;
        end;

    [global]
    procedure print_profiles (file_name : file_name_string);
        var
            h, n, k, sum, number_of_entries : integer;
            name : name_string;

        procedure integer_to_string (i : integer; var s : name_string);
            var
                j : integer;
            begin
            for j := 1 to max_name do
                s[j] := ' ';
            j := max_name;
            while (i > 0) and (j >= 1) do
                begin
                n := i mod 10;
                i := i div 10;
                s[j] := chr(n + ord('0'));
                j := j - 1;
                end;
            end;

        begin
        open (profile_output, file_name, history := new);
        rewrite (profile_output);
        for h := 1 to max_heuristics do
            begin
            number_of_entries := 0;
            for n := 0 to max_n do
                begin
                for k := 0 to max_k do
                    if frequency[h, n, k] <> 0 then
                        number_of_entries := number_of_entries + 1;
                end;
            if number_of_entries > 0 then
                begin
                integer_to_string (h, name);
                writeln (profile_output, name, '      ',
                    h:5, number_of_entries:10);
                for n := 0 to max_n do
                    begin
                    sum := 0;
                    for k := 0 to max_k do
                        sum := sum + frequency[h, n, k];
```

AD-A173 096 AN EMPIRICAL STUDY IN THE SIMULATION OF HEURISTIC ERROR 4/4
BEHAVIOR(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB
OH S R HANSEN 1986 AFIT/CI/NR-86-184T

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
1.1
1.25
1.4
1.6

```
                    for k := 0 to max_k do
                        if frequency[h, n, k] <> 0 then
                            writeln (profile_output, n:5, k:5,
                                ((frequency[h, n, k]*100)/sum):6:1,
                                frequency[h, n, k]:10);
                    end;
                end;
            end;
        close (profile_output);
        end;

(*
 *  heuristic calculation routines
 *)

    function tile_position (tile : char; goal : puzzle_state) : integer;
        var
            i : integer;
        begin
        i := center;
        while (goal[i] <> tile) do
            i := i + 1;
        tile_position := i;
        end;

    function perimeter_distance (tile1_pos, tile2_pos : integer) : integer;
        var
            d : integer;
        begin
        d := abs (tile1_pos - tile2_pos);
        if (d > (num_positions - 1 - d)) then
            d := num_positions - 1 - d;
        perimeter_distance := d;
        end;

    function center_distance (tile_pos : integer) : integer;
        var
            d, i : integer;
        begin
        if tile_pos = center then
            center_distance := 0
        else
            begin
            d := num_positions;
            for i := first to num_positions do
                if link[i] and (perimeter_distance (i, tile_pos) < d) then
                    d := perimeter_distance (i, tile_pos);
            center_distance := d + 1;
            end;
        end;

    function tiles_misplaced (current, goal : puzzle_state) : integer;
        var
        i, n : integer;
    begin
    n := 0;
    for i := center to num_positions do
        if (current[i] <> goal[i]) and (current[i] <> chr(0)) then
            n := n + 1;
    tiles_misplaced := n;
    end;

function manhattan_distance (current, goal : puzzle_state) : integer;
    var
        m_dist, i, j, d, d2 : integer;
    begin
    m_dist := 0;
    for i := center to num_positions do
        begin
        j := tile_position (current[i], goal);
        if (i <> j) and (current[i] <> chr(0)) then
            begin
            if (i = center) then
                d := center_distance (j)
            else if (j = center) then
                d := center_distance (i)
            else
                begin
                d := center_distance (j) + center_distance (i);
                d2 := perimeter_distance (i, j);
                if (d2 < d) then
                    d := d2;
                end;
            m_dist := m_dist + d;
            end;
        end;
    manhattan_distance := m_dist;
    end;

function enhanced_manhattan_distance (current, goal : puzzle_state) :
            integer;
    var
        i, j, next_i, next_j, score : integer;
    begin
    score := 0;
    for i := first to num_positions do
        begin
        if (current[i] <> chr(0)) then
            begin
            j := tile_position (current[i], goal);
            if (j <> center) then
                begin
                if (j = num_positions) then
                    next_j := first
                else
                    next_j := j + 1;
                if (i = num_positions) then
                      next_i := first
                else
                     next_i := i + 1;
                if (current[next_i] <> goal[next_j])
                    and (current[next_i] <> chr(0))
                    and (goal[next_i] <> chr(0)) then
                        score := score + 2;
                end;
            end;
        end;
    enhanced_manhattan_distance :=
            manhattan_distance (current, goal) + 3 * score;
    end;

function simulated_by_histogram (n : integer;
                                 p : profile_pointer) : integer;
    var
        j, k, accum : integer;
    begin
    with p^ do
        begin
        k := min[n];
        accum := histogram[n, k];
        j := random_integer_between (1, count[n]);
        while (j > accum) do
            begin
            k := k + 1;
            accum := accum + histogram[n, k];
            end;
        end;
    simulated_by_histogram := k;
    end;

function simulated_by_distribution (n : integer;
                                    p : profile_pointer;
                                    d : distribution_type) : integer;
    var
        i : integer;
        r : real;
    begin
    with p^ do
        begin
        r := random_by_distribution (d);
        i := round ((stdev[n] * r) + mean[n]);
        if (i < 0) then
            i := 0;
        end;
    simulated_by_distribution := i;
    end;

function worst_case_by_profile (current : node_ptr;
                                n, c_star : integer;
```

```
                                    p : profile_pointer) : integer;
    var
        k : integer;
    begin
    with p^ do
        if ((current^.g_value + n) > c_star) then
            k := round (mean[n] - 2 * stdev[n])
        else
            k := round (mean[n] + 2 * stdev[n]);
    if k < 0 then
        k := 0;
    worst_case_by_profile := k;
    end;

function proportional_error (n : integer;
                             r, l : real;
                             d : distribution_type) : integer;
    var
        b : real;
    begin
    b := random_by_distribution (d);
    proportional_error := round (n * (1 + l + b*(r - l)));
    end;


[global]
function estimated_distance (heuristic : integer;
                             current   : node_ptr;
                             goal      : puzzle_state;
                             c_star    : integer) : integer;
    var
        n, k : integer;
        inv_current : node_ptr;

    begin
    if current^.state = goal then
        k := 0
    else
        begin
          if (inv_graph^.state <> goal) then
            begin
              free_binary_tree (inv_search_tree);
              initialize_graph_descriptor (inv_g);
              generate_graph (goal, inv_search_tree,
                              inv_graph, inv_g);
            end;
        inv_current := find_state_in_tree (current^.state,
                                           inv_search_tree);
        n := inv_current^.g_value;

        case heuristic of
            1 : k := tiles_misplaced (current^.state, goal);
            2 : k := manhattan_distance (current^.state, goal);
```

```
                3 : k := enhanced_manhattan_distance (current^.state, goal);

            4,5,6 : k := simulated_by_distribution (n, profile[heuristic-3],
                                                    normal);
            7,8,9 : k := simulated_by_histogram (n, profile[heuristic-6]);
            10,11,
              12  : k := worst_case_by_profile (current, n, c_star,
                                                profile[heuristic-9]);

               13 : k := proportional_error (n, -0.5, -1.0, linear);
               14 : k := proportional_error (n, 0.0, -0.7, linear);
               15 : k := proportional_error (n, 0.0, -0.5, linear);
               16 : k := proportional_error (n, 0.5, -0.2, linear);
               17 : k := proportional_error (n, -0.5, -1.0, nonlinear);
               18 : k := proportional_error (n, 0.0, -0.7, nonlinear);
               19 : k := proportional_error (n, 0.0, -0.5, nonlinear);
               20 : k := proportional_error (n, 0.5, -0.2, nonlinear);
                end;

            end;
        frequency[heuristic, n, k] := frequency[heuristic, n, k] + 1;
        estimated_distance := k;
        end;

end.
```

```
(*
 *       Statistics Module  (v4.0  23-Feb-86  AJC/SRH)
 *)

module statistics (input, output, distribution_file);
    const
        max_pairs = 100;
        max_name = 10;
        max_file_name = 30;
        seed = 4489;

    type
        distribution_index = 1..max_pairs;
        name_string = packed array [1..max_name] of char;
        file_name_string = packed array [1..max_file_name] of char;

        distribution_type = (normal, linear, nonlinear);

        distribution_pointer = ^distribution_record;
        distribution_record = record
                name : distribution_type;
                pairs : distribution_index;
                abscissa, ordinate : array [distribution_index] of real;
                end;

    var
        random_seed : integer;
        distribution : array [distribution_type] of distribution_pointer;
        distribution_file : text;

    [global]
    procedure initialize_statistics;
        begin
        random_seed := seed;
        distribution[normal] := nil;
        distribution[linear] := nil;
        distribution[nonlinear] := nil;
        end;

    [external, asynchronous]
    function mth$random (var seed : integer) : real;
        extern;

    [global]
    function random_integer_between (m, n : integer) : integer;
        var
            p : real;
            q : integer;
        begin
        p := mth$random (random_seed);
        q := round (p * (n - m)) + m;
        if (m <= q) and (q <= n) then
            random_integer_between := q
        else
            begin
            writeln ('Warning -- random out of bounds');
            halt;
            end;
        end;

(*
 *  Distribution functions
 *)

    procedure create_distribution (var d : distribution_pointer);
        var
            i : integer;
        begin
        new (d);
        with d^ do
            begin
            pairs := 0;
            for i := 1 to max_pairs do
                begin
                abscissa[i] := 0.0;
                ordinate[i] := 0.0;
                end;
            end;
        end;

    [global]
    procedure read_distributions (file_name : file_name_string);
        var
            d : distribution_pointer;
            i : integer;
        begin
        open (distribution_file, file_name, history := old);
        reset (distribution_file);
        while not eof (distribution_file) do
            begin
            create_distribution (d);
            with d^ do
                begin
                readln (distribution_file, name, pairs);
                for i := 1 to pairs do
                    readln (distribution_file, abscissa[i], ordinate[i]);
                end;
            distribution[d^.name] := d;
            end;
        close (distribution_file);
        end;

    [global]
    function random_by_distribution (d : distribution_type) : real;
    var
        i : integer;
        x : real;
        done : boolean;
    begin
    x := mth$random (random_seed);
    if distribution[d] <> nil then
        with distribution[d]^ do
            begin
            i := 1;
            done := false;
            while (i <= pairs) and (not done) do
                begin
                if x <= ordinate[i] then
                    done := true
                else
                    i := i + 1;
                end;
            if done then
                random_by_distribution := abscissa[i]
            else
                begin
                writeln (' *** Error: Random not found in distribution');
                halt;
                end;
            end;
    end;

end.
```

```
*************************************************************************
*                                                                       *
*                              APPENDIX B                               *
*                                                                       *
*                  Beads World Tools Definition Modules                 *
*                                                                       *
*************************************************************************
```

```
(*
 *        Utilities Definition Module  (v4.0  23-Feb-86  AJC/SRH)
 *)

(*
 *  From UTILITIES Import CONST
 *)
        max_positions = 10;
        no_puzzle = '          ';
        center = 1;
        first = 2;

(*
 *  From UTILITIES Import TYPE
 *)
        positions = center..max_positions;
        puzzle_state = packed array [positions] of char;

        node_ptr = ^puzzle_node;

        neighbor_node_ptr = ^neighbor_node;
        neighbor_node = record
                neighbor : node_ptr;
                next : neighbor_node_ptr;
                end;

        puzzle_node = record
                state : puzzle_state;
                left, right, sort_left, sort_right : node_ptr;
                neighbors : neighbor_node_ptr;
                parent : node_ptr;
                g_value, h_value : integer;
                f_value : real;
                end;

(*
 *  From UTILITIES Import:
 *)

    [external]
    procedure read_state (var s : puzzle_state; n : integer);
        external;
```

```
[external]
procedure print_state (s : puzzle_state; n : integer);
    external;

[external]
function create_puzzle_node (s : puzzle_state): node_ptr;
    external;

[external]
procedure free_node (var n : node_ptr);
    external;

[external]
procedure create_empty_list (var list : node_ptr);
    external;

[external]
function is_empty (list : node_ptr) : boolean;
    external;

[external]
procedure place_on_end_of_list (p, list : node_ptr);
    external;

[external]
procedure place_in_ascending_order (p, list : node_ptr);
    external;

[external]
function remove_from_front_of_list (list : node_ptr) : node_ptr;
    external;

[external]
procedure delete_from_list (p, list : node_ptr);
    external;

[external]
procedure free_list (var list : node_ptr);
    external;

[external]
procedure insert_in_tree (n : node_ptr; var tree : node_ptr);
    external;

[external]
function find_in_tree (n, tree : node_ptr) : node_ptr;
    external;

[external]
function find_state_in_tree (s : puzzle_state;
                             tree : node_ptr) : node_ptr;
    external;
```

```
[external]
procedure free_binary_tree (var t : node_ptr);
    external;

[external]
procedure free_graph (var g : node_ptr);
    external;
```

```
(*
 *       Control Definition Module  (v4.0  23-Feb-86  AJC/SRH)
 *)

(*
 *  From CONTROL Import CONST
 *)
      max_links = max_positions;
      max_levels = 99;

(*
 *  From CONTROL Import TYPE
 *)
      link_array = array[first..max_positions] of boolean;

      level_record = record
              count : integer;
              list : node_ptr;
              end;

      level_array = array [0..max_levels] of level_record;

      graph_descriptor = record
              depth, generated, expanded : integer;
              level : level_array;
              end;

      results_descriptor = record
              solved : boolean;
              path_length, min_path_length, generated,
                  expanded, heuristic : integer;
              weight : real;
              start, goal : node_ptr;
              end;

(*
 *  From CONTROL Import VAR
 *)
      num_positions, num_links : [global] integer;
      link : [global] link_array;

(*
 *  From CONTROL Import:
 *)

   [external]
   procedure initialize_control (np, nl : integer; l : link_array);
       external;

   [external]
   procedure initialize_graph_descriptor (var g : graph_descriptor);
```

```
    external;

[external]
procedure generate_graph (start_state : puzzle_state;
                          var search_tree, graph : node_ptr;
                          var g : graph_descriptor);
    external;

[external]
procedure initialize_results (var r : results_descriptor);
    external;

[external]
procedure print_puzzle_solution (r : results_descriptor);
    external;

[external]
procedure ordered_search (start, goal : puzzle_state;
                          heuristic : integer;
                          weight : real;
                          var results : results_descriptor);
    external;

[external]
procedure graph_search (start, goal : puzzle_state;
                        heuristic : integer;
                        weight : real;
                        var results : results_descriptor);
    external;
```

```
(*
 *       Heuristics Definitiion Module  (v4.0  23-Feb-86  AJC/SRH)
 *)

(*
 *  From HEURISTIC Import CONST
 *)
      max_heuristics = 24;
      max_file_name = 30;

(*
 *  From HEURISTIC Import TYPE
 *)
      file_name_string = packed array [1..max_file_name] of char;

(*
 *  From HEURISTIC Import:
 *)

  [external]
  procedure read_profiles (file_name : file_name_string);
      external;

  [external]
  procedure print_profiles (file_name : file_name_string);
      external;

  [external]
  procedure initialize_heuristics;
      external;

  [external]
  function estimated_distance (heuristic : integer;
                               current   : node_ptr;
                               goal      : puzzle_state;
                               c_star    : integer) : integer;
      external;
```

```
(*
 *       Statistics Definition Module  (v4.0  23-Feb-86  AJC/SRH)
 *)

(*
 *  From STATISTICS Import CONST
 *)
        max_file_name = 30;

(*
 *  From STATISTICS Import TYPE
 *)
        file_name_string = packed array [1..max_file_name] of char;

        distribution_type = (normal, linear, nonlinear);

(*
 *  From STATISTICS Import:
 *)

    [external]
    procedure initialize_statistics;
        external;

    [external]
    function random_integer_between (m, n : integer) : integer;
        external;

    [external]
    procedure read_distributions (file_name : file_name_string);
        external;

    [external]
    function random_by_distribution (d : distribution_type) : real;
        external;
```

```
*********************************************************************
*                                                                   *
*                             APPENDIX C                            *
*                                                                   *
*                   Beads World Applications Modules                *
*                                                                   *
*********************************************************************
```

```
(*
 *        Application GENERATE_GRAPH  (v4.0  26-Feb-86  AJC/SRH)
 *)

program generate_graph (input, output);
    const
(*
 *  From UTILITIES Import CONST
 *)
        max_positions = 10;
        no_puzzle = '          ';
        center = 1;
        first = 2;

(*
 *  From CONTROL Import CONST
 *)
        max_links = max_positions;
        max_levels = 99;

    type
(*
 *  From UTILITIES Import TYPE
 *)
        positions = center..max_positions;
        puzzle_state = packed array [positions] of char;

        node_ptr = ^puzzle_node;

        neighbor_node_ptr = ^neighbor_node;
        neighbor_node = record
                neighbor : node_ptr;
                next : neighbor_node_ptr;
                end;

        puzzle_node = record
                state : puzzle_state;
                left, right, sort_left, sort_right : node_ptr;
                neighbors : neighbor_node_ptr;
                parent : node_ptr;
                g_value, h_value : integer;
                f_value : real;
                end;
```

```
(*
 *  From CONTROL Import TYPE
 *)
        link_array = array [first..max_positions] of boolean;

        level_record = record
                count : integer;
                list : node_ptr;
                end;

        level_array = array [0..max_levels] of level_record;

        graph_descriptor = record
                depth, generated, expanded : integer;
                level : level_array;
                end;

    var
        link : link_array;
        num_positions, num_links : integer;

        start : puzzle_state;
        search_tree, graph : node_ptr;
        problem, i, num, opcode, min_sample : integer;
        gd : graph_descriptor;

(*
 *  From UTILITIES Import:
 *)

    [external]
    procedure read_state (var s : puzzle_state; n : integer);
        external;

    [external]
    procedure print_state (s : puzzle_state; n : integer);
        external;

    [external]
    procedure free_binary_tree (var t : node_ptr);
        external;

    [external]
    procedure free_graph (var g : node_ptr);
        external;

(*
 *  From CONTROL Import:
 *)

    [external]
    procedure initialize_control (np, nl : integer; l : link_array);
```

```
        external;

    [external]
    procedure initialize_graph_descriptor (var g : graph_descriptor);
        external;

    [external]
    procedure generate_graph (start_state : puzzle_state;
                              var search_tree, graph : node_ptr;
                              var g : graph_descriptor);
        external;

(*
 *  From STATISTICS Import:
 *)

    [external]
    function random_integer_between (m, n : integer): integer;
        external;


(*
 *  GENERATE_GRAPH
 *)

(*
 *  destructive print of puzzle states in graph
 *)

    procedure print_graph_space (g : node_ptr);
        var
            n : neighbor_node_ptr;
        begin
        if g^.state <> no_puzzle then
            begin
            print_state (g^.state, num_positions);
            writeln;
            g^.state := no_puzzle;
            n := g^.neighbors;
            while n <> nil do
                begin
                print_graph_space (n^.neighbor);
                n := n^.next;
                end;
            end;
        end;

    procedure report_graph_statistics (problem : integer;
                                       g : graph_descriptor);
        var
            avg_neighbors, tot, lev : real;
            i, j, dups, spots : integer;
        begin
        writeln;
        writeln;
        writeln;
        writeln (' GRAPH STATISTICS : # ', problem:2);
        writeln;
        writeln ('   Positions :  ', num_positions:2);
        write ('   Links     :  ');
        for i := first to num_positions do
            if link[i] then
                write (i:3);
        writeln;
        writeln;
        write ('   Starting Configuration :  ');
        print_state (start, num_positions);
        writeln;
        writeln;
        writeln ('   Nodes Generated    :  ', g.generated:5);
        writeln ('   Nodes Expanded     :  ', g.expanded:5);
        avg_neighbors := (g.generated - 1) / g.expanded;
        writeln ('   Avg # of Neighbors :  ', avg_neighbors:5:2);
        dups := g.generated - g.expanded;
        writeln ('   Number of "dupes"  :  ', dups:5);
        writeln;
        writeln ('   Depth    = ', g.depth:7);
        writeln;
        writeln ('   Nodes at each level: ');
        writeln;
        tot := g.expanded;
        for i := 0 to g.depth do
            begin
            write ('     ', i:2, ' -- ', g.level[i].count:5, '  ');
            lev := g.level[i].count;
            spots := round(100 * (lev / tot));
            for j := 1 to spots do
                write ('*');
            writeln;
            end;
        writeln;
        writeln;
        end;

(*
 * Sample generation routines
 *)

    function find_ith_member (i : integer; list : node_ptr) : node_ptr;
        var
            k : integer;
            p : node_ptr;
        begin
        p := list^.right;
        k := 1;
        while (p <> list) and (k < i) do
            begin
            p := p^.right;
            k := k + 1;
            end;
        if p <> list then
            find_ith_member := p
        else
            find_ith_member := nil;
        end;

    procedure generate_sample (g : graph_descriptor);
        var
            start, q : node_ptr;
            i, j, r, sample_size : integer;
            lev, tot : real;
        begin
        write (num_positions:4, num_links:4);
        for i := first to num_positions do
            if link[i] then
                write (i:3);
        writeln;
        start := find_ith_member (1, g.level[0].list);
        tot := g.expanded;
        for i := 1 to g.depth do
            begin
            lev := g.level[i].count;
            sample_size := round ((lev * 100) / tot) + min_sample;
            if g.level[i].count < sample_size then
                sample_size := g.level[i].count;
            for j := 1 to sample_size do
                begin
                repeat
                    begin
                    r := random_integer_between (1, trunc(lev));
                    q := find_ith_member (r, g.level[i].list);
                    end
                until q^.g_value <> 0;
                q^.g_value := 0;
                write (i:4);
                print_state (start^.state, num_positions);
                print_state (q^.state, num_positions);
                writeln;
                end;
            end;
        end;


(*
 * main program
 *)

    begin
```

```
problem := 1;
while not eof do
    begin
    for i := first to max_positions do
        link[i] := false;
    read (num_positions, num_links);
    for i := 1 to num_links do
        begin
        read (num);
        link[num] := true;
        end;
    initialize_control (num_positions, num_links, link);
    read (opcode);
    if (opcode = 1) then
        read (min_sample);
    readln;
    read_state (start, num_positions);
    readln;
    initialize_graph_descriptor (gd);
    generate_graph (start, search_tree, graph, gd);
    if (opcode = 0) then
        report_graph_statistics (problem, gd)
    else if (opcode = 1) then
        generate_sample (gd)
    else if (opcode = 2) then
        print_graph_space (graph);
    free_graph (graph);
    problem := problem + 1;
    end;
end.
```

```
(*
 *       Application SOLVE  (v4.0  23-Feb-86 AJC/SRH)
 *)

program solve (input, output);
    const
(*
 *  From UTILITIES Import CONST
 *)
        max_positions = 10;
        no_puzzle = '          ';
        center = 1;
        first = 2;

(*
 *  From CONTROL Import CONST
 *)
        max_links = max_positions;
        max_levels = 99;

(*
 *  From HEURISTIC Import CONST
 *)
        max_heuristics = 24;
        max_file_name = 30;

(*
 *  SOLVE
 *)
        no_file_name = '                              ';
        max_integer = 999999999;
        max_weights = 11;

    type
(*
 *  From UTILITIES Import TYPE
 *)
        positions = center..max_positions;
        puzzle_state = packed array [positions] of char;

        node_ptr = ^puzzle_node;

        neighbor_node_ptr = ^neighbor_node;
        neighbor_node = record
                neighbor : node_ptr;
                next : neighbor_node_ptr;
                end;

        puzzle_node = record
                state : puzzle_state;
                left, right, sort_left, sort_right : node_ptr;
                neighbors : neighbor_node_ptr;
                parent : node_ptr;
                g_value, h_value : integer;
                f_value : real;
                end;

(*
 *  From CONTROL Import TYPE
 *)
        link_array = array [first..max_links] of boolean;

        results_descriptor = record
                solved : boolean;
                path_length, min_path_length, generated,
                    expanded, heuristic : integer;
                weight : real;
                start, goal : node_ptr;
                end;

(*
 *  From HEURISTIC Import TYPE
 *)
        file_name_string = packed array [1..max_file_name] of char;

(*
 *  SOLVE
 *)
        heuristic_array = array [1..max_heuristics] of integer;
        weight_array    = array [1..max_weights] of real;

        aggregate_array = array [1..max_heuristics, 1..max_weights] of
                                        integer;
        aggregate_statistics = record
                xmin, xmax, lmin, lmax,
                xtotal, ltotal, xmean, lmean : aggregate_array;
                end;
        aggregate_stats_ptr = ^aggregate_statistics;

        search_method_types = (ordered, graph);

    var
(*
 *  SOLVE
 *)
        num_positions, num_links : integer;
        link : link_array;

        search_method : search_method_types;

        start, goal : puzzle_state;
        results : results_descriptor;

        profile_input, profile_output,
                distribution_input : file_name_string;

        i, j, k, l, num, number_of_heuristics,
            number_of_weights, opcode : integer;
        heuristic : heuristic_array;
        weight : weight_array;
        a : aggregate_stats_ptr;
        old_n, no_n, n : integer;

(*
 *  From UTILITIES Import:
 *)

    [external]
    procedure read_state (var s : puzzle_state; n : integer);
        external;

    [external]
    procedure print_state (s : puzzle_state; n : integer);
        external;

    [external]
    procedure free_binary_tree (var t : node_ptr);
        external;

(*
 *  From CONTROL Import:
 *)

    [external]
    procedure initialize_control (np, nl : integer; l : link_array);
        external;

    [external]
    procedure initialize_results (var r : results_descriptor);
        external;

    [external]
    procedure print_puzzle_solution (r : results_descriptor);
        external;

    [external]
    procedure ordered_search (start, goal : puzzle_state;
                              heuristic : integer;
                              weight : real;
                              var results : results_descriptor);
        external;

    [external]
    procedure graph_search (start, goal : puzzle_state;
                            heuristic : integer;
                            weight : real;
                            var results : results_descriptor);
        external;

(*
 *  From HEURISTIC Import:
 *)

    [external]
    procedure initialize_heuristics;
        external;

    [external]
    procedure read_profiles (file_name : file_name_string);
        external;

    [external]
    procedure print_profiles (file_name : file_name_string);
        external;

(*
 *  From STATISTIC Import:
 *)

    [external]
    procedure read_distributions (file_name : file_name_string);
        external;

(*
 *  SOLVER
 *)

    procedure report_puzzle_results (r : results_descriptor);
        begin
        writeln;
        writeln;
        writeln (' PROBLEM SOLUTION RESULTS ', r.heuristic:4, r.weight:5:2);
        writeln;
        if not r.solved then
            writeln ('   No solution found! ');
        writeln;
        write ('   Start: ');
        print_state (r.start^.state, num_positions);
        writeln;
        write ('   Goal:  ');
        print_state (r.goal^.state, num_positions);
        writeln;
        writeln;
        writeln ('   Nodes Generated     :  ', r.generated:5);
        writeln ('   Nodes Expanded      :  ', r.expanded:5);
        writeln ('   Path Length         :  ', r.path_length:5);
        writeln ('   Minimum Path Length :  ', r.min_path_length:5);
        writeln;
        writeln;
        end;
```

```
procedure generate_data (r : results_descriptor);
    begin
    writeln (r.heuristic:5,
             r.weight:5:2,
             r.min_path_length:5,
             r.path_length:5,
             r.generated:5,
             r.expanded:5);
    end;

procedure init_aggregate_results (a : aggregate_stats_ptr);
    var
        k, l : integer;
    begin
    for k := 1 to number_of_heuristics do
        for l := 1 to number_of_weights do
            begin
            with a^ do
                begin
                xmin[k][l] := max_integer;
                xmean[k][l] := 0;
                xmax[k][l] := 0;
                lmin[k][l] := max_integer;
                lmean[k][l] := 0;
                lmax[k][l] := 0;
                xtotal[k][l] := 0;
                ltotal[k][l] := 0;
                end;
            end;
    end;

procedure print_aggregate_results (old_n : integer;
                    number_of_heuristics : integer;
                               heuristic : heuristic_array;
                       number_of_weights : integer;
                              aggregates : aggregate_stats_ptr);
    var
        i, j : integer;
    begin
    with aggregates^ do
        begin
    for i := 1 to number_of_heuristics do
        begin
        write (old_n:3);
        write (heuristic[i]:3);
        for j := 1 to number_of_weights do
            write (xmin[i][j]:6);
        writeln;
        write (old_n:3);
        write (heuristic[i]:3);
        for j := 1 to number_of_weights do
```

```
                write (xmean[i][j]:6);
            writeln;
            write (old_n:3);
            write (heuristic[i]:3);
            for j := 1 to number_of_weights do
                write (xmax[i][j]:6);
            writeln;
            write (old_n:3);
            write (heuristic[i]:3);
            for j := 1 to number_of_weights do
                write (lmin[i][j]:6);
            writeln;
            write (old_n:3);
            write (heuristic[i]:3);
            for j := 1 to number_of_weights do
                write (lmean[i][j]:6);
            writeln;
            write (old_n:3);
            write (heuristic[i]:3);
            for j := 1 to number_of_weights do
                write (lmax[i][j]:6);
            writeln;
            end;
            end;
        end;

(*
 * problem solution routines
 *)

    procedure solve (search_method : search_method_types);
        begin
        for i := 1 to number_of_heuristics do
            for j := 1 to number_of_weights do
                begin
                if search_method = ordered then
                    ordered_search (start, goal, heuristic[i],
                                    weight[j], results)
                else
                    graph_search (start, goal, heuristic[i],
                                    weight[j], results);
                report_puzzle_results (results);
                free_binary_tree (results.start);
                end;
        end;

    procedure solve_and_print (search_method : search_method_types);
        begin
        for i := 1 to number_of_heuristics do
            for j := 1 to number_of_weights do
                begin
                if search_method = ordered then
                    ordered_search (start, goal, heuristic[i],
                                weight[j], results)
                else
                    graph_search (start, goal, heuristic[i],
                                  weight[j], results);
                report_puzzle_results (results);
                print_puzzle_solution (results);
                free_binary_tree (results.start);
                end;
        end;

    procedure solve_and_generate (search_method : search_method_types);
        begin
        for i := 1 to number_of_heuristics do
            for j := 1 to number_of_weights do
                begin
                if search_method = ordered then
                    ordered_search (start, goal, heuristic[i],
                                    weight[j], results)
                else
                    graph_search (start, goal, heuristic[i],
                                  weight[j], results);
                generate_data (results);
                free_binary_tree (results.start);
                end;
        end;

    procedure solve_and_aggregate (search_method : search_method_types);
        begin
        if (n <> old_n) and (old_n <> 0) then
            begin
            print_aggregate_results (old_n,
                                     number_of_heuristics,
                                     heuristic,
                                     number_of_weights,
                                     a);
            init_aggregate_results (a);
            no_n := 1;
            end;
        old_n := n;
        for i := 1 to number_of_heuristics do
            for j := 1 to number_of_weights do
                begin
                if search_method = ordered then
                    ordered_search (start, goal, heuristic[i],
                                    weight[j], results)
                else
                    graph_search (start, goal, heuristic[i],
                                  weight[j], results);
                with a^ do
                    begin
                    if results.expanded < xmin[i][j] then
                            xmin[i][j] := results.expanded;
                    xtotal[i][j] := xtotal[i][j] + results.expanded;
                    xmean[i][j] := round(xtotal[i][j] / no_n);
                    if results.expanded > xmax[i][j] then
                            xmax[i][j] := results.expanded;
                    if results.path_length < lmin[i][j] then
                            lmin[i][j] := results.path_length;
                    ltotal[i][j] := ltotal[i][j] + results.path_length;
                    lmean[i][j] := round(ltotal[i][j] / no_n);
                    if results.path_length > lmax[i][j] then
                            lmax[i][j] := results.path_length;
                    end;
                free_binary_tree (results.start);
                end;
        end;

    procedure extract_file_name (var f : file_name_string;
                                delimiter : char);
        var
            i : integer;
            ch : char;
        begin
        for i := 1 to max_file_name do
            f[i] := ' ';
        ch := ' ';
        while (not eoln) and (ch <> delimiter) do
            read (ch);
        if ch = delimiter then
            begin
            read (ch);
            i := 1;
            while (not eoln) and (ch <> delimiter)
                    and (i <= max_file_name) do
                begin
                f[i] := ch;
                i := i + 1;
                read (ch);
                end;
            end;
        end;

(*
 *  Main program
 *)

    begin
    for i := first to max_positions do
        link[i] := false;
    read (num_positions, num_links);
    for i := 1 to num_links do
        begin
        read (num);
        link[num] := true;
        end;
    initialize_control (num_positions, num_links, link);
```

```
readln (opcode, search_method);
read (number_of_heuristics);
for i := 1 to number_of_heuristics do
    read (heuristic[i]);
read (number_of_weights);
for i := 1 to number_of_weights do
    read (weight[i]);
readln;
extract_file_name (profile_input, '"');
extract_file_name (profile_output, '"');
readln;
extract_file_name (distribution_input, '"');
readln;
initialize_results (results);
if opcode < 2 then
    begin
    writeln;
    write ('   Positions  :  ', num_positions:3);
    write ('   Links      :  ');
    for i := first to num_positions do
        if link[i] then
            write (i:3);
    writeln;
    write ('   Heuristics :  ', number_of_heuristics:3);
    for i := 1 to number_of_heuristics do
        write (heuristic[i]:3);
    writeln;
    write ('   Weights :  ', number_of_weights:3);
    for i := 1 to number_of_weights do
        write (weight[i]:5:2);
    writeln;
    end
else
    begin
    write (num_positions:3, num_links:3);
    for i := first to num_positions do
        if link[i] then
            write (i:3);
    writeln;
    write (number_of_heuristics:3);
    for i := 1 to number_of_heuristics do
        write (heuristic[i]:3);
    write (number_of_weights:6);
    for i := 1 to number_of_weights do
        write (weight[i]:6:2);
    end;
writeln;

if profile_input <> no_file_name then
    read_profiles (profile_input);
if distribution_input <> no_file_name then
    read_distributions (distribution_input);
```

```
if opcode >= 3 then
    begin
    new (a);
    init_aggregate_results (a);
    old_n := 0;
    no_n := 0;
    end;

while (not eof) do
    begin
    read (n);
    no_n := no_n + 1;
    results.min_path_length := n;
    read_state (goal, num_positions);
    read_state (start, num_positions);
    readln;
    case opcode of
        0 : solve (search_method);
        1 : solve_and_print (search_method);
        2 : solve_and_generate (search_method);
        3 : solve_and_aggregate (search_method);
        end;
    end;

if opcode >= 3 then
    begin
    print_aggregate_results (old_n,
                             number_of_heuristics,
                             heuristic,
                             number_of_weights,
                             a);
    dispose (a);
    end;

if profile_output <> no_file_name then
    print_profiles (profile_output);
end.
```

```
*-------------------------------------------------------------------*
*                                                                   *
*        Application GRAFER  (v4.0  6-Mar-86  AJC/SRH)              *
*                                                                   *
*-------------------------------------------------------------------*

      PROGRAM GRAFER

      integer crvparms (12,3), heur (15), doagain
      real wt (11)
      dimension datafl (25,50,6,11), pkedlns (200)
*                      (k, n, type, w)
      character*5 types(6)
      data (types (i), i=1,6) /'XMin','XMean','XMax','LMin','LMean',
     +               'LMax'/

      call readdata (datafl, maxn, maxheur, heur, wt, maxwt)

 23   call menu (maxheur, heur, maxwt, wt, types, ncurves, crvparms,
     +               maxn, option)

      call plottype()

      call setaxis (ncurves, crvparms, types,
     +                       pkedlns, wt, maxwt, maxn, option)

      call drawcrvs (heur, wt, maxwt, types, ncurves, crvparms,
     +                      datafl, pkedlns, maxn, option)

      call height (.10)
      if (option .eq. 1) then
              call legend (pkedlns, ncurves + 2, .2, 2.35)
      elseif ((option .eq. 2) .or. (option .eq. 4)) then
              call legend (pkedlns, ncurves + 1, .2, 2.25)
      else
              call legend (pkedlns, ncurves, .2, 2.5)
              endif

      call endpl (0)

      if (doagain () .eq. 1) then
              goto 23
              endif

      call donepl
      stop
      end

*********************************************************************
      subroutine menu (maxheur, heur, maxwt, wt, types, ncurves,
     +               crvparms, maxn, option)
```

```
*************************************************************************
      integer heur (15), crvparms (12, 3)
      real wt (11)
      character*5 types (6)
      integer gettype, getheur, getwt, doagain

      ncurves = 0
1198  print *, 'Do you want :'
      print *, '      1:  X vrs N graph'
      print *, '      2:  L vrs N graph'
      print *, '      3:  X vrs W graph'
      print *, '      4:  L vrs W graph'
      read *, option
      if ((option .lt. 1) .or. (option .gt. 4)) then
              print *, 'invalid response'
              go to 1198
              endif
      if (option .eq. 3) then
              i = getheur (maxheur, heur)
              do  3001 j= 1, 5
                      ncurves = ncurves + 1
                      crvparms (ncurves, 1) = 2
                      crvparms (ncurves, 2) = i
                      crvparms (ncurves, 3) = j * 4
3001          continue
      elseif (option .eq. 4) then
              i = getheur (maxheur, heur)
              do  3002 j= 1, 5
                      ncurves = ncurves + 1
                      crvparms (ncurves, 1) = 5
                      crvparms (ncurves, 2) = i
                      crvparms (ncurves, 3) = j * 4
3002          continue
      else
1199          print *, 'Do you want:'
              print *, '      1:  All Actual K, one weight, one type'
              print *, '      2:  All types, one weight, one K'
              print *, '      3:  Selected W, one K, one type'
              print *, '      4:  All W, one K, one type'
              print *, '      5:  Actual K vrs Simulated Ks,',
     +                                   ' one wt, one type'
              print *, '      6:  Other variation graph'
              read *, ians
              if ((ians .lt. 1) .or. (ians .gt. 6)) then
                      print *, ' Invalid response'
                      go to 1199
                      endif
              if (ians .eq. 1) then
                      j = gettype (types, option)
                      i = getwt (maxwt, wt)
                      do 185 k=1, 3
                              crvparms (k, 1) = j
                              crvparms (k, 2) = k
                              crvparms (k, 3) = i
185                   continue
                      ncurves = 3
              elseif (ians .eq. 2) then
                      i = getheur (maxheur, heur)
                      j = getwt (maxwt, wt)
                      i1 = option * 3
                      k2 = 3
                      do 186 k1 = 1, 3
                              crvparms (k2, 1) = i1 - k1 + 1
                              crvparms (k2, 2) = i
                              crvparms (k2, 3) = j
                              k2 = k2 - 1
186                   continue
                      ncurves = 3
              elseif (ians .eq. 3) then
                      i = getheur (maxheur, heur)
                      j = gettype (types, option)
                      do 1871 k=1, 4
                              crvparms (k, 1) = j
                              crvparms (k, 2) = i
1871                  continue
                      crvparms (1, 3) = 1
                      crvparms (2, 3) = 2
                      crvparms (3, 3) = 4
                      crvparms (4, 3) = 7
                      ncurves = 4
              elseif (ians .eq. 4) then
                      i = getheur (maxheur, heur)
                      j = gettype (types, option)
                      do 1872 k=1, maxwt
                              crvparms (k, 1) = j
                              crvparms (k, 2) = i
                              crvparms (k, 3) = k
1872                  continue
                      ncurves = maxwt
              elseif (ians .eq. 5) then
                      i = getheur (maxheur, heur)
                      j = gettype (types, option)
                      l = getwt (maxwt, wt)
                      do 188 k=1, maxheur / 3
                              crvparms (k, 1) = j
                              crvparms (k, 2) = i + 3*(k-1)
                              crvparms (k, 3) = l
188                   continue
                      ncurves = maxheur / 3
              else
                      again = 1
200                   if (again .eq. 1) then
                          ncurves = ncurves + 1
                          crvparms (ncurves, 1) = gettype (types, option)
                          crvparms (ncurves, 2) = getheur (maxheur, heur)
```

```
                           crvparms (ncurves, 3) = getwt (maxwt, wt)
                           again = dosgain ()
                           go to 200
                        endif
                   endif
              endif
              return
              end


*************************************************************************
            subroutine setaxis (ncurves, crvparms, types, pkedlns, wt,
     +                         maxwt, maxn, option)
*************************************************************************
      integer crvparms (12, 3), itext (4)
      dimension pkedlns (200)
      real wt (11)
      character*5 types (6)
      character*16 title
      dimension datafl(25,50,6,11), xarray(25), yarray(25), ybrfrst(20)
*                        (k, n, type, w)
      data (ybrfrst (i), i=1, 20) /1, 5, 14, 23, 44, 56, 113, 173,
     +        263, 377, 592, 932, 1265, 1651, 2014, 2254, 2342,
     +        2493, 2514, 2519/

      xdimen = 4.0
      ydimen = 3.0
      call reset ('all')
      call triplx
      call height (.17)
      call area2d (xdimen, ydimen)
      call sclpic (.8)
      call dot
      if (option .eq. 1) then
          call graf (0.0, 20.0, 20.0, 1.0, 1.0, ydimen)
          call height (0.08)
          call xgraxs (0.0, 0.1, 1.0, xdimen, ' $', -100, 0.0, 0.0)
          call xticks (5)
          call height (.17)
          call xintax
          call xgraxs (0.0, 5.0, float(maxn), xdimen,
     +              'Depth of Goal (N)$', 100, 0.0, 0.0)
          call yaxang (0.0)
          call ylgaxs (1.0, 0.75, ydimen,
     +              'Nodes Expanded (X)$', 100, 0.0, 0.0)
          call height (.10)
          call lines ('optimal$', pkedlns, 1)
          call lines ('breadth first$', pkedlns, 2)
          ientry = 2
          do 50 i=1, maxn
              xarray (i) = float (i)
              yarray (i) = float (i)
 50       continue
```

```
            call marker (15)
            call curve (xarray, yarray, maxn, 1)
            call marker (18)
            call curve (xarray, ybrfrst, 20, 1)
        elseif (option .eq. 2) then
            call yname ('Path Length (L)$', 100)
            call yticks (2)
            call yaxang (0.0)
            call yintax
            print *, 'input L axis length:'
            read  *, laxis
            call graf (0.0, 5.0, float(maxn), 0.0, 1.0, laxis)
            call height (0.08)
            call xgraxs (0.0, 0.1, 1.0, xdimen, ' $', -100, 0.0, 0.0)
            call xticks (5)
            call height (0.17)
            call xintax
            call xgraxs (0.0, 5.0, float(maxn), xdimen,
     +                'Depth of Goal (N)$', 100, 0.0, 0.0)
            call height (.10)
            call lines ('optimal$', pkedlns, 1)
            ientry = 1
            do 51 i=1, maxn
                xarray (i) = float (i)
                yarray (i) = 1.0
51          continue
            call marker (15)
            call curve (xarray, yarray, maxn, 1)
        elseif (option .eq. 3) then
            call graf (0.0, 20.0, 20.0, 1.0, 1.0, ydimen)
            call xgraxs (0.0, 0.1, 1.0, xdimen, 'Weight$', 100, 0.0, 0.0)
            call yaxang (0.0)
            call ylgaxs (1.0, 0.75, ydimen,
     +                'Nodes Expanded (X)$', 100, 0.0, 0.0)
            call height (.10)
            ientry = 0
        elseif (option .eq. 4) then
            call xname ('Weight$',100)
            call yname ('Path Length (L)$', 100)
            call yticks (5)
            call yaxang (0.0)
            call yintax
            call graf (0.0, 0.1, 1.0, 0.0, 1.0, 5.0)
            call height (.10)
            call lines ('optimal$', pkedlns, 1)
            ientry = 1
            do 52 i=1, maxwt
                xarray (i) = wt(i)
                yarray (i) = 1.0
52          continue
            call marker (15)
            call curve (xarray, yarray, maxwt, 1)
            endif
        call reset ('dot')

*** NOW LABEL THE LINES ***
        do 100 i=1, ncurves
            j1 = crvparms (i, 1)
            j2 = crvparms (i, 3)
            title = types (j1)
            j3 = index (title, ' ')
            if ((option .eq. 1) .or. (option .eq. 2)) then
                write (title (j3:), fmt=111) crvparms (i, 2), wt (j2)
 111            format (',K' , I2 , ',' , F4.2 , '$' )
                read (title, '(4A4)') itext
                call lines (itext, pkedlns, i + ientry)
            else
                write (title (j3:), fmt=112) crvparms (i, 2), j2
 112            format (',K' , I2 , ',N=' , I2 , '$' )
                read (title, '(4A4)') itext
                call lines (itext, pkedlns, i + ientry)
                endif
 100    continue
        return
        end

*************************************************************************
        subroutine drawcrvs (heur, wt, maxwt, types, ncurves,
     +                        crvparms, datafl, pkedlns, maxn, option)
*************************************************************************
        integer heur (15), crvparms (12,3)
        real wt (11), datafl(25,50,6,11), xarray (25), yarray (25)
        character*5 types (6)
        dimension pkedlns (200)

**loop for each entry in crvparms
        markit = 14
        if ((option .eq. 1) .or. (option .eq. 2)) then
                do 301  j=1, ncurves
                        type = crvparms (j, 1)
                        h = crvparms (j, 2)
                        w = crvparms (j, 3)
                        do 300  i=1, maxn
                            xarray (i) = float (i)
                            if (option .eq. 1) then
                                 yarray(i) = datafl (h, i, type, w)
                            elseif (option .eq. 2) then
                                 yarray(i) = datafl (h, i, type, w)
     +                                                / float (i)
                            endif
 300                    continue
                        call marker (markit)
                        call curve (xarray, yarray, maxn, 1)
                        markit = markit - 1
 301            continue
        else
```

```
                do 330  j=1, ncurves
                        type = crvparms (j, 1)
                        h = crvparms (j, 2)
                        n = crvparms (j, 3)
                        do 331 i=1, maxwt
                            xarray(i) = wt(i)
                            if (option .eq. 3) then
                                yarray(i) = datafl(h, n, type, i)
                            else
                                yarray(i) = datafl(h, n, type, i) / n
                                endif
331                     continue
                        call marker (markit)
                        call curve (xarray, yarray, maxwt, 1)
                        markit = markit - 1
330              continue
        endif
        return
        end

************************************************************************
        subroutine readdata (datafl, maxn, maxheur, heur, wt, maxwt)
************************************************************************
        dimension datafl(25,50,6,11)
        real wt (11)
        integer heur (15)

****** load data file array with aggregate values ***************
        open (1, file='search.out',status='old',err=105)
*****        scan past first values (get weights and num of wts)
        read (1,*,end=105) nodes, nlinks, (links, i=1,nlinks)
        read (1,*,end=105) maxheur, (heur (i), i=1,maxheur),
     +        maxwt, (wt (i), i=1, maxwt)

 100    read (1,*,end=110)
     +        (n, k, (datafl (k,n,ln,i), i=1, maxwt), ln=1, 6)
        go to 100
 110    close (1)
        maxn = n
        go to 199
***** abnormal file condition
 105    print *, 'error with input file'
        stop
 199    end

************************************************************************
        subroutine plottype ()
************************************************************************

        print *,'Do you want plot       1: On Terminal'
        print *,'                            2: Printed'
        read *,ians
        if (ians .eq. 2) then
```

```
                call qms
                print *,'remember to type @LASER when done'
        else
                print *,'SET TERMINAL UP FOR TEKTRONICS MODE NOW'
                print *,'       Type 1 when ready'
                read *, answer
                call tekall(4010,480,0,0,0)
        endif
        return
        end

**************************************************************
        integer function gettype(types, option)
**************************************************************
        character*5 types(6)

        if (option .eq. 1) then
            min = 1
            max = 3
        else
            min = 4
            max = 6
            endif
 1120   print *,'specify a plot type:'
        write (*,fmt=1121) (i, types (i), i=min,max)
 1121   format (10x,I1,':  ',A5,' vrs N')
        read *, answer
        if ((answer .lt. min) .or. (answer .gt. max)) then
                print *,' Invalid option'
                go to 1120
                endif
        gettype = answer
        return
        end

***********************************************************
        integer function getheur (maxheur, heur)
***********************************************************
        integer heur (15)

 1125   print *,'select a heuristic'
        write (*, fmt=1126) (i, heur (i), i=1, maxheur)
 1126   format (10X, I2, ':  K', I2)
        read *, answer
        if ((answer .lt. 1) .or. (answer .gt. maxheur)) then
                print *,'Invalid option'
                go to 1125
                endif
        getheur = heur(answer)
        return
        end

***********************************************************
```

```
        integer function getwt (maxwt, wt)
************************************************************
        real wt (11)

        if (maxwt .gt. 0) then
 1130           print *,'select the weight'
                write (*, fmt=5) (i, wt(i), i=1, maxwt)
   5            format (10x, I2, ':  ', F3.1)
                read *, answer
                if ((answer .lt. 1) .or. (answer .gt. maxwt)) then
                        print *,'Invalid option'
                        go to 1130
                        endif
        else
                answer = 0
                endif
        getwt = answer
        return
        end

************************************************************
        integer function doagain
************************************************************

 1140   print *,'do you want to put another curve on this plot?'
        print *,'          0: No'
        print *,'          1: Yes'
        read *, answer
        if ((answer .lt. 0) .or. (answer .gt. 1)) then
                print *,'Invalid option'
                go to 1140
                endif
        doagain = answer
        return
        end
```

```
*-----------------------------------------------------------------------*
*                                                                       *
*          Application GRAFPROF  (v4.0  6-Mar-86  AJC/SRH)              *
*                                                                       *
*-----------------------------------------------------------------------*

      PROGRAM GRAFPROF

      integer crvparms (12,2), heur (25)
      real wt (11)
      dimension pkedlns (200)
      dimension profile (25,50,4)
      dimension histo (25,50,75), percents (25,50,75)
      character*5 types(3)
      data (types (i), i=1,3) /'KMin','KMean','KMax'/

      call readprof (profile, percents, histo, heur,
     +                      maxheur, maxn, typeprof)

11    call menu (maxheur, heur, types, ncurves, crvparms,
     +               profile, maxn, option)

      call plottype()

      if (option .eq. 3) then
              iheur = crvparms(1,2)
              call draw3d (profile, percents, histo,
     +                             iheur, maxn, typeprof)
      else
              call setaxis (maxn, typeprof)
              call drawcrvs (heur, types, ncurves, crvparms,
     +                      pkedlns, maxn, profile, option)
              if (option .eq. 4) then
                      call endpl (0)
                      iheur = crvparms(1,2)
                      call draw3d (profile, percents, histo,
     +                                     iheur, maxn, typeprof)
                      endif
              endif

      call endpl (0)

      print *, 'Do another?  0: No'
      print *, '             1: Yes'
      read  *, ians
      if (ians .eq. 1) then
          go to 11
      else
          call donepl
          stop
          endif
```

```
        end

************************************************************************
        subroutine menu (maxheur, heur, types, ncurves,
     +                  crvparms, profile, maxn, option)

************************************************************************

        dimension profile(25, 50, 4)
        integer heur(25), crvparms(12, 2)
        character*5 types(3)
        integer gettype, getheur, getwt, doagain

        ncurves = 0
        maxwt = 0

 1199   print *, 'Do you want:'
        print *, '      1:  All K, one type'
        print *, '      2:  One K, all types, incl standard deviation'
        print *, '      3:  3-D Profile for one K'
        print *, '      4:  Full set for one K (options 2 and 3)'
        print *, '      5:  Other variation graph'
        read *, option
        if ((option .lt. 1) .or. (option .gt. 5)) then
                print *, ' Invalid response'
                go to 1199
                endif
        if (option .eq. 1) then
                j = gettype (types)
                do 185 k=1, maxheur
                        crvparms (k, 1) = j
                        crvparms (k, 2) = heur(k)
 185            continue
                ncurves = maxheur
        elseif ((option .eq. 2) .or. (option .eq. 4)) then
                i = getheur (maxheur, heur)
                do 186 k = 1, 3
                        crvparms (k, 1) = k
                        crvparms (k, 2) = i
 186            continue
                ncurves = 3
        elseif (option .eq. 3) then
                crvparms (1,2) = getheur (maxheur, heur)
        else
            again = 1
 200        if (again .eq. 1) then
                ncurves = ncurves + 1
                crvparms (ncurves, 1) = gettype (types)
                crvparms (ncurves, 2) = getheur (maxheur, heur)
                again = doagain ()
                go to 200
                endif
        endif
```

```
      return
      end

************************************************************************
      subroutine readprof (profile, percents, histo,
     +                            heur, maxheur, maxn, typeprof)
************************************************************************
      dimension profile (25,50,4)
      integer heur (25)
      dimension histo (25,50,75), percents (25,50,75)
      real ksum, k2sum, nsum
      character*15 char
      maxheur = 25
      maxn = 50
      maxk = 75

      do 21 i=1,maxheur
         heur(i) = 0
         do 21 j=1, maxn
           do 21 k=1, maxk
              histo (i,j,k) = 0.0
              percents (i,j,k) = 0.0
 21   continue

 22   print *, 'Do you want    1:  Source profiles'
      print *, '               2:  Run profiles'
      read *, typeprof
      if ((typeprof .lt. 1) .or. (typeprof .gt. 2)) then
              print *, 'Invalid option'
              go to 22
              endif
      if (typeprof .eq. 1) then
              open (2, file='profile.pro',status='old',err=105)
      else
              open (2, file='profile.run',status='old',err=105)
              endif

 99   read (2,*,end=111) char, heuristic, entries

      maxn = 0
      do 102 i=1,entries
          read (2,*,end=111) n, k, percent, count
          histo (heuristic, n+1, k+1) = count
          percents (heuristic, n+1, k+1) = percent / 100.0
          heur(heuristic) = heuristic
          if (n .gt. maxn) then
              maxn = n
              endif
 102  continue
      go to 99
***** abnormal file condition
 105  print *, 'error with input file'
      stop
```

```
111    close (2)
       j = 0
       do 67 i=1, maxheur
          if (heur(i) .gt. 0) then
               j = i
               endif
67     continue
       maxheur = j

       do 66 i=1, maxheur
           do 66 j=1, maxn + 1
               profile (i,j,1) = 9999.0
               profile (i,j,2) = 0.0
               profile (i,j,3) = 0.0
               profile (i,j,4) = 0.0
               nsum = 0
               ksum = 0
               k2sum = 0
               do 65 k=1, maxk
                   if (histo (i,j,k) .ne. 0) then
                       nsum = nsum + histo (i,j,k)
                       ksum = ksum + (k-1) * histo (i,j,k)
                       k2sum = k2sum + (k-1) * (k-1) * histo (i,j,k)
                       if ((k-1) .lt. profile(i,j,1)) then
                           profile(i,j,1) = k - 1
                           endif
                       if ((k-1) .gt. profile(i,j,3)) then
                           profile(i,j,3) = k - 1
                           endif
                       endif
65             continue
               if (nsum .gt. 0) then
                   profile(i,j,2) = ksum / nsum
               else
                   profile(i,j,1) = 0
                   endif
               if (nsum .gt. 1) then
                   profile(i,j,4) =
     +                 sqrt(abs(ksum*ksum/nsum - k2sum)/(nsum-1))
                  endif
66     continue
       return
       end

*************************************************************************
       subroutine plottype ()
*************************************************************************

       print *,'Do you want plot        1: On Terminal'
       print *,'                        2: Printed'
       read *,ians
       if (ians .eq. 2) then
```

```
              call qms
              print *,'remember to (laser filename) when done'
        else
              print *,'SET TERMINAL UP FOR TEKTRONICS MODE NOW'
              print *,'   Enter 1 when ready:'
              read *, ians
              call tekall(4010,480,0,0,0)
        endif
        return
        end

**************************************************************************
        subroutine setaxis (maxn, typeprof)
**************************************************************************

        call reset ('all')
        call triplx
        call height (.17)
        xdimen = 4.0
        ydimen = 3.0
        call area2d (xdimen, ydimen)
        call head2 (typeprof)
        call sclpic (.8)
        call dot
        call marker (13)
        call yname ('Est Dist (K)$', 100)
        call yticks (5)
        call yaxang (0.0)
        call yintax
        call graf (0.0, 5.0, float(maxn), 0.0, 5.0, 50.0)
        call height (0.08)
        call xgraxs (0.0, 0.1, 1.0, xdimen, ' $', -100, 0.0, 0.0)
        call xticks (5)
        call height (0.17)
        call xintax
        call xgraxs (0.0, 5.0, float(maxn), xdimen,
     +        'True Dist (i)$', 100, 0.0, 0.0)
        return
        end

**************************************************************************
        subroutine drawcrvs (heur, types, ncurves,
     +                 crvparms, pkedlns, maxn, profile, option)
**************************************************************************
        integer heur(25), crvparms (12,2), itext(4)
        real  xarray (50), yarray (50)
        character*5 types (3)
        character*16 title
        dimension pkedlns (200), profile (25, 50, 4)

        call height (.10)
        call lines ('optimal$', pkedlns, 1)
        do 52 i=1, maxn + 1
```

```
                xarray (i) = float (i - 1)
                yarray (i) = float (i - 1)
 52     continue
        call curve (xarray, yarray, maxn + 1, 1)
        call reset ('dot')

        do 100 i=1, ncurves
            j1 = crvparms (i, 1)
 100    continue

** for each entry in crvparms
        markit = 15
        do 301  j=1, ncurves
**   set up name for legend
                type = crvparms (j, 1)
                title = types (type)
                j3 = index (title, ' ')
                write (title (j3:), fmt=112) crvparms (j, 2)
 112            format (',K' , I2 , '$')
                read (title, '(4A4)') itext
                call lines (itext, pkedlns, j + 1)
                heuristic = crvparms (j, 2)
**    get data for the curve and draw it
                do 300  i=1, maxn + 1
                    xarray (i) = float (i - 1)
                    yarray (i) = profile (heuristic, i, type)
 300            continue
                call marker (markit)
                call curve (xarray, yarray, maxn + 1, 1)
                markit = markit + 1
 301    continue
        if ((option .eq. 2) .or. (option .eq. 4)) then
            call lines('+/- 1 Std. Dev.$',pkedlns, ncurves + 2)
            do 302 i=1, maxn + 1
                yarray(i) = profile (heuristic, i, 2)
     +                    + profile (heuristic, i, 4)
 302        continue
            call dot
            call marker(markit)
            call curve (xarray, yarray, maxn + 1, 1)
            do 303 i=1, maxn + 1
                yarray(i) = profile (heuristic, i, 2)
     +                    - profile (heuristic, i, 4)
 303        continue
            call marker (markit)
            call curve (xarray, yarray, maxn + 1, 1)
            endif

        call height (.10)
        if ((option .eq. 2) .or. (option .eq. 4)) then
            call legend (pkedlns, ncurves + 2, .2, 2.25)
        else
            call legend (pkedlns, ncurves + 1, .2, 2.25)
        endif

        return
        end

*************************************************************
        integer function gettype(types)
**************************************************************
        character*5 types(3)

 1120   print *,'specify a plot type:'
        write (*,fmt=1121) (i, types (i), i=1,3)
 1121   format (10x,I1,':  ',A5,' vrs N')
        read *, answer
        if ((answer .lt. 1) .or. (answer .gt. 3)) then
                print *,' Invalid option'
                go to 1120
                endif
        gettype = answer
        return
        end

**********************************************************
        integer function getheur (maxheur, heur)
**********************************************************
        integer heur(25)

 1125   print *,'select a heuristic'
        do 1126 i=1, 15
            if (heur(i) .ne. 0) then
                write (*, fmt=1127) i, heur(i)
                endif
 1126   continue
 1127   format (10X, I2, ':  K', I2)
        read *, answer
        if ((answer .lt. 1)
     +           .or. (answer .gt. maxheur)
     +           .or. (heur(answer) .eq. 0)) then
                print *,'Invalid option'
                go to 1125
                endif
        getheur = answer
        return
        end

**********************************************************
        integer function doagain
**********************************************************

 1140   print *,'do you want to put another curve on this plot?'
        print *,'        0: No'
        print *,'        1: Yes'
```

```
      read *, answer
      if ((answer .lt. 0) .or. (answer .gt. 1)) then
              print *,'Invalid option'
              go to 1140
              endif
      doagain = answer
      return
      end


************************************************************************
      subroutine draw3d (profile, percents, histo, iheur,
     +                              maxn, typeprof)
************************************************************************
      integer  heur (25)
      dimension profile (25,50,4)
      dimension histo (25,50,75), percents (25,50,75)
      dimension zarray (21,51)
      dimension xray(21), yray(51), zray(51)

      call reset ('all')
      call triplx
      call height (.17)
      call area2d (4.0, 3.5)
      call head1 (iheur, typeprof)
      call sclpic (.8)
      call marker (13)
      call height (0.10)
      call x3name ('True Dist (i)$', 100)
      call y3name ('Est Dist (K)$', 100)
      call z3name ('Freq(%)$',100)
*** set the perspective point for the graph ***
      print *,'input X perspective:'
      read *,xx
      print *, 'input Y perspective:'
      read *, yy
      print *, 'input Z perspective:'
      read *, zz
      call vuabs (xx, yy, zz)
      call volm3d (4.0, 4.0, 1.0)
      call yticks (5)
      call xticks (5)
      call zaxang (-90.0)
      call xintax
      call yintax
      call graf3d (0.0, 5.0, float(maxn),
     +             0.0, 5.0, 50.0,
     +             0.0, 0.25, 1.0)
      do 11 n=1, maxn + 1
        do 11 k=1, 51
            zarray(n,k) = percents(iheur, n, k)
 11   continue
      call surmat(zarray, 1, maxn + 1, 1, 51, 0)
      call grfiti (0.0,4.0,0.0, 4.0,4.0,0.0, 0.0,4.0,1.0)
      call area2d (4.0,1.0)
      call yaxang (0.0)
      call graf (0.0,5.0,float(maxn), 0.0,0.25,1.0)

      do 111 n=1, maxn + 1
          xray (n) = n - 1
          yray (n) = 0.0
          do 111 k=1, 51
              if (percents(iheur, n, k) .gt. yray(n)) then
                  yray(n) = percents(iheur, n, k)
                  endif
111   continue
      call curve (xray, yray, maxn + 1, 1)
      call dot
      call grid (1,1)
      call reset('dot')
      call height (0.17)
      return
      end


***********************************************************************
      subroutine head1 (iheur, typeprof)
***********************************************************************
      integer  itext(6)
      character*24 title

      if (typeprof .eq. 1) then
          title = 'Source Profile, K'
          write (title (18:), fmt=112) iheur
      else
          title = 'Run Profile, K'
          write (title (15:), fmt=112) iheur
          endif
112   format ( I2 , '$')
      read (title, '(6A4)') itext
      call headin (itext, 100, 1.5, 1)
      return
      end


***********************************************************************
      subroutine head2 (typeprof)
***********************************************************************

      if (typeprof .eq. 1) then
          call headin ('Source Profile$', 100, 1.5, 1)
      else
          call headin ('Run Profile$', 100, 1.5, 1)
          endif
      return
```

end

APPENDIX D

Distribution Disk Contents

General Beads World Utilities Modules:

UTILITIES.PAS	--	Utilities module
CONTROL.PAS	--	Control structures module
HEURBAS.PAS	--	Basic heuristics module
STATISTIC.PAS	--	Statistics module

Beads World Utilities Definition Files:

UTILITIES.DEF	--	Utilities definitions
CONTROL.DEF	--	Control structure definitions
HEURISTIC.DEF	--	Heuristic definitions
STATISTIC.DEF	--	Statistic definitions

Applications Modules:

GRAPH.PAS	--	Beads World graph generator
SOLVE.PAS	--	A* puzzle solver
HEURMOD.PAS	--	Additional heuristics
GRAFER.FOR	--	Graphics package (solutions)
GRAFPROF.FOR	--	Graphics package (profiles)

APPENDIX E

KEY TO GRAPH SYMBOLS AND TERMS

<u>Symbol</u> <u>Meaning</u>

MIN- Minimum of all the values observed in the sample.

MAX- Maximum of all the values observed in the sample.

MEAN- The average of all the values in the sample.

N- Depth or level in the state space.

I- Actual minimal distance of a node from the goal.

X- Number of nodes expanded.

L- Normalized solution path length.

W- The weight used in the Weighted A* algorithm.

K- The estimate provided by the heuristic of I.

BIBLIOGRAPHY

Gaschnig, J., "Performance Measurement and Analysis of Certain Search Algorithms". PhD Dissertation. Carnegie-Mellon University. 1979.

Lizza, C., "Generation of Flight Paths Using Heuristic Search". Wright State University. 1985.

Newell, A., J. Shaw, and H.A. Simon, "Empirical Explorations with the LOGIC THEORY Machine: A Case Study in Heuristics," in Computers and Thought (E. Feigenbaum and J. Feldman, eds.). McGraw-Hill Book Co. New York. 1963.

Nilsson, N., Principles of Artificial Intelligence. Tioga Press. Palo Alto, Calif. 1980.

Pearl, J., Heuristics: Intelligent Strategies for Computer Problem Solving. Addison-Wesley. 1984.

Pohl, I., "First Results on the Effect of Error in Heuristic Search," Machine Intelligence 5, B. Meltzer and D. Michie (eds.). Edinburgh University Press. Edinburgh. 1970.

END

12-86

DTIC